

25th International Conference on Types for Proofs and Programs

TYPES 2019, June 11–14, 2019, Oslo, Norway

Edited by

Marc Bezem

Assia Mahboubi



Editors

Marc Bezem

University of Bergen, Norway
Marc.Bezem@uib.no

Assia Mahboubi

Inria, Nantes, France
Vrije Universiteit Amsterdam, The Netherlands
assia.mahboubi@inria.fr

ACM Classification 2012

Theory of computation → Type theory

ISBN 978-3-95977-158-0

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-158-0>.

Publication date

September, 2020

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0):
<https://creativecommons.org/licenses/by/3.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2019.0

ISBN 978-3-95977-158-0

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

LIPICs – Leibniz International Proceedings in Informatics

LIPICs is a series of high-quality conference proceedings across all fields in informatics. LIPICs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Christel Baier (TU Dresden)
- Mikolaj Bojanczyk (University of Warsaw)
- Roberto Di Cosmo (INRIA and University Paris Diderot)
- Javier Esparza (TU München)
- Meena Mahajan (Institute of Mathematical Sciences)
- Dieter van Melkebeek (University of Wisconsin-Madison)
- Anca Muscholl (University Bordeaux)
- Luke Ong (University of Oxford)
- Catuscia Palamidessi (INRIA)
- Thomas Schwentick (TU Dortmund)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Marc Bezem and Assia Mahboubi</i>	0:vii
List of Authors	
.....	0:ix
Regular Papers	
Making Isabelle Content Accessible in Knowledge Representation Formats	
<i>Michael Kohlhase, Florian Rabe, and Makarius Wenzel</i>	1:1–1:24
Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules	
<i>Jesper Cockx</i>	2:1–2:27
A Quantitative Understanding of Pattern Matching	
<i>Sandra Alves, Delia Kesner, and Daniel Ventura</i>	3:1–3:36
Big Step Normalisation for Type Theory	
<i>Thorsten Altenkirch and Colin Geniet</i>	4:1–4:20
From Cubes to Twisted Cubes via Graph Morphisms in Type Theory	
<i>Gun Pinyo and Nicolai Kraus</i>	5:1–5:18
For Finitary Induction-Induction, Induction Is Enough	
<i>Ambrus Kaposi, András Kovács, and Ambroise Lafont</i>	6:1–6:30
Eta-Equivalence in Core Dependent Haskell	
<i>Anastasiya Kravchuk-Kirilyuk, Antoine Voizard, and Stephanie Weirich</i>	7:1–7:31
Coherence for Monoidal Groupoids in HoTT	
<i>Stefano Piceghello</i>	8:1–8:20
Is Impredicativity Implicitly Implicit?	
<i>Stefan Monnier and Nathaniel Bos</i>	9:1–9:19
Higher Inductive Type Elimimators Without Paths	
<i>Nils Anders Danielsson</i>	10:1–10:18



■ Preface

This volume constitutes the post-proceedings of the *25th International Conference on Types for Proofs and Programs, TYPES 2019*, held in Oslo, Norway, 11–14 June 2019.

The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalised and computer assisted reasoning and computer programming. The meetings from 1990 to 2008 were annual workshops of a sequence of five EU-funded networking projects. Since 2009, TYPES has been run as an independent conference series. Previous TYPES meetings were held in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal near Nijmegen (2002), Torino (2003), Jouy-en-Josas near Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Torino (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014), Tallinn (2015), Novi Sad (2016), Budapest (2017), and Braga (2018).

The TYPES areas of interest include, but are not limited to: foundations of type theory and constructive mathematics; applications of type theory; dependently typed programming; industrial uses of type theory technology; meta-theoretic studies of type systems; proof assistants and proof technology; automation in computer-assisted reasoning; links between type theory and functional programming; formalizing mathematics using type theory.

The TYPES conferences are of open and informal character. Selection of contributed talks is based on short abstracts; reporting work in progress and work presented or published elsewhere is welcome. A formal post-proceedings volume is prepared after the conference; papers submitted to that volume must represent unpublished work and are subjected to a full peer-review process.

TYPES 2019 was held in parallel with HoTT-UF, the workshop on Homotopy Type Theory and Univalent Foundations, 12–14 June 2019, in Oslo. Wednesday 12 June the two events had a joint programme. Both events were part of the Special Year 2018/19 on Homotopy Type Theory and Univalent Foundations at the Centre for Advanced Study (CAS) at the Norwegian Academy of Science and Letters.

The program of the conference consisted of 50 contributed short presentations and four invited lectures of one hour. The invited lecturers were: Adam Chlipala, Conor McBride, Assia Mahboubi and Stephanie Weirich. The combined events TYPES 2019 and HoTT-UF gathered 115 participants from around 20 countries.

There were 12 submissions to this open post-proceedings volume, the large majority related to presentations at the conference. After a thorough peer-review procedure of two rounds, 10 submissions could be accepted for publication. We thank all authors, reviewers, and members of the program committee for their contribution to this volume.

Sponsors

The Centre for Advanced Study (CAS) at the Norwegian Academy of Science and Letters provided generous support, both financial and administrative, which we gratefully acknowledge. We are also grateful for the support of COST Action CA15123 EUTypes and of the Research Council of Norway, project 240810 Computational Aspects of Univalence (2015–2020).

Marc Bezem and Assia Mahboubi, July 2020

25th International Conference on Types for Proofs and Programs (TYPES 2019).
Editors: Marc Bezem and Assia Mahboubi




Leibniz International Proceedings in Informatics
LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ List of Authors


Thorsten Altenkirch (4)
School for Computer Science,
University of Nottingham, UK

Sandra Alves (3)
DCC-FCUP & CRACS,
University of Porto, Portugal

Nathaniel Bos (9)
McGill University – SOCS, Montréal, Canada

Jesper Cockx  (2)
Department of Software Technology,
TU Delft, The Netherlands

Nils Anders Danielsson  (10)
University of Gothenburg, Sweden


Colin Geniet  (4)
Computer Science Department,
ENS Paris-Saclay, France

Ambrus Kaposi  (6)
Eötvös Loránd University, Budapest, Hungary


Delia Kesner (3)
Université de Paris, CNRS, IRIF, France
Institut Universitaire de France, France

Michael Kohlhase  (1)
University Erlangen-Nürnberg, Germany


András Kovács  (6)
Eötvös Loránd University, Budapest, Hungary


Nicolai Kraus  (5)
School of Computer Science,
University of Nottingham, UK

Anastasiya Kravchuk-Kirilyuk (7)
Princeton University, NJ, USA

Ambroise Lafont  (6)
IMT Atlantique, Inria, LS2N CNRS,
Nantes, France

Stefan Monnier  (9)
Université de Montréal – DIRO, Canada


Stefano Piceghello  (8)
Department of Informatics and Department of
Mathematics, University of Bergen, Norway

Gun Pinyo  (5)
School of Computer Science,
University of Nottingham, UK

Florian Rabe (1)
University Erlangen-Nürnberg, Germany

Daniel Ventura (3)
INF, Universidade Federal de Goiás,
Goiânia, Brazil

Antoine Voizard (7)
University of Pennsylvania,
Philadelphia, PA, USA

Stephanie Weirich  (7)
University of Pennsylvania,
Philadelphia, PA, USA

Makarius Wenzel (1)
Selfemployed, Augsburg, Germany



Making Isabelle Content Accessible in Knowledge Representation Formats

Michael Kohlhase 

University Erlangen-Nürnberg, Germany

<https://kwarc.info/kohlhase>

Michael.Kohlhase@fau.de

Florian Rabe

University Erlangen-Nürnberg, Germany

Makarius Wenzel

Selfemployed, Augsburg, Germany

<https://sketis.net/>

Abstract

The libraries of proof assistants like Isabelle, Coq, HOL are notoriously difficult to interpret by external tools: de facto, only the prover itself can parse and process them adequately. In the case of Isabelle, an export of the library into a FAIR (Findable, Accessible, Interoperable, and Reusable) knowledge exchange format was already envisioned by the authors in 1999 but had previously proved too difficult.

After substantial improvements of the Isabelle Prover IDE (PIDE) and the OMDoc/MMT format since then, we are now able to deliver such an export. Concretely we present an integration of PIDE and MMT that allows exporting all Isabelle libraries in OMDoc format. Our export covers the full Isabelle distribution and the Archive of Formal Proofs (AFP) – more than 12 thousand theories and locales resulting in over 65 GB of OMDoc/XML.

Such a systematic export of Isabelle content to a well-defined interchange format like OMDoc enables many applications such as dependency management, independent proof checking, or library search.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Isabelle, PIDE, OMDoc, MMT, library, export

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.1

Supplementary Material The translated libraries are available at <https://gl.mathhub.info/Isabelle> as compressed OMDoc files.

Funding The authors were supported by DFG grant RA-18723-1 OAF and EU grant Horizon 2020 ERI 676541 OpenDreamKit.

1 Introduction and Related Work

Motivation. A critical bottleneck in the field of interactive theorem proving is the lack of interoperability between proof assistants and related tools. This leads to a duplication of efforts: both formalizations and auxiliary tool support (e.g., for automated proving, library management, user interfaces) cannot be easily shared between systems. This situation is well-understood by the community and has persisted for decades despite occasional attempts to achieve interoperability by standardization or library translations.

The story of this article started in 1999, when one author (Kohlhase, who worked on the OMDoc interchange format [27] for formal libraries) wrote an email to another one (Wenzel, who worked on the Isabelle proof assistant [43, 44]) asking about the status of ongoing efforts to export Isabelle theories in some format that could be further transformed into OMDoc. Just 19 years later, Wenzel replied to the same email announcing that an Isabelle→OMDoc export now works routinely. Critically, this export was enabled by the PIDE and MMT infrastructures developed for Isabelle by Wenzel resp. for OMDoc by Rabe in the interim.



© Michael Kohlhase, Florian Rabe, and Makarius Wenzel;

licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 1; pp. 1:1–1:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Despite this massive groundwork laid in the last two decades, the export itself still required about 9 person-months to implement. This paper tells the story of how we achieved this export after such a long time.

Isabelle99 (October 1999) was a rather small experimental proof assistant for multiple object logics, with ≈ 1 MB source text for Isabelle/ZF library and ≈ 3 MB for Isabelle/HOL. The ZF library was particularly interesting for Kohlhase at that time and considered large. In contrast, Isabelle2020 (April 2020) includes ≈ 2 MB material for ZF and ≈ 30 MB for HOL, or rather ≈ 160 MB if the Archive of Formal Proofs (AFP) is included. The PIDE/MMT work flow described in this paper requires a server-class machine to handle all this material: 80 GB RAM, 8 CPU cores, and 22 h elapsed time (this includes theory and proof processing by Isabelle). Thus, a major portion of publicly known Isabelle content¹ becomes accessible as XML in the OMDOC format: 65 GB uncompressed or 300 MB with XZ compression.

Related Work. In both formalizations and auxiliary tool support, previous work has shown significant potential for knowledge sharing. Regarding sharing among proof assistants, library translations such as [41, 23, 26, 34] have been used to transport theorems across systems. An unusual approach is virtualization of HOL4 in Isabelle [18], where the ML environment of Isabelle is carefully instrumented to load the HOL4 library sources (also in ML) and reconstruct theories and proofs within the Isabelle/Pure inference kernel.

Most of these approaches produce an isolated image of the source library within the target library. Alignments [21] have been used to match pragmatically corresponding concepts defined in different libraries [10]. In contrast, [18] connects interesting results via *lifting and transfer*, where only the signatures of the main conclusions need to be taken into account.

Regarding sharing among proof assistants and auxiliary tools, Isabelle/Sledgehammer [35, 44] integrates different automation tools generically, and Dedukti [7] has been used as independent checker for various proof assistant libraries. Premise selection tools use, e.g., machine-learning [22], to reduce the search space when running automated provers on subgoals. In all cases, a single tool could be used for every proof assistant – provided the language and library are available in a universal format that can be plugged into it.

Unfortunately, the latter point – the universal format – is often prohibitively expensive for many interesting applications. Firstly, it is extremely difficult to design a format that strikes a good trade-off between simplicity and universality. And secondly, even in the presence of such a format, it is difficult to implement the export of a library into it. Here it is important to realize that any export attempt is doomed that uses a custom parser or type checker for the library – only the internal data structures maintained by the proof assistant are informative enough for most use cases. Consequently, only expert developers can perform this step, and of these, each proof assistant community only has very few.

In previous work, the authors have developed such a universal format [27, 48, 29] for formal knowledge: OMDOC is an XML language geared towards making formula structure and context dependencies explicit while remaining independent of the underlying logical formalism. We also built a strong implementation – the MMT system – and a number of generic services, e.g., [46, 30]. In the DFG-funded OAF Project (Open Archive of Formalization), we have developed export for Mizar [17], HOL Light [24], IMPS [6], PVS [28], and Coq in [38]. In what we now call the *OAF approach*, we systematically

- (i) defined the logic of the proof assistant in a logical framework by hand,
- (ii) instrumented the proof assistant to export its libraries, and
- (iii) use the instrumented prover to export the libraries

¹ In the Isabelle community, contributions are usually submitted to AFP for long-term maintenance, and thus become centrally accessible. Only a few exceptional projects are maintained independently (e.g. seL4 <https://sel4.systems> or IsaFoR <http://cl-informatik.uibk.ac.at/isafor>).

for all these exports. MMT provides the semantics that ties together the three involved levels (logical framework, logic, and library) and provides a uniform high-level API for further processing. [32] gives an overview over the theoretical, technical, and social challenges of the OAF exports.

In the work reported here, we follow this basic recipe with a few modifications. Firstly, because Isabelle already includes a logical framework, we do not encode Isabelle in yet another one. Instead, we extend the existing LF formalization in MMT to obtain one for the Pure framework of Isabelle. There are two reasons for this choice: it is conceptually appropriate as it puts the logics defined in Isabelle on the same levels as those defined in other logical frameworks (e.g., MMT/LF/HOL Light and MMT/Isabelle/HOL); it also improves scalability by avoiding another layer of logical framework-encoding. Secondly, Isabelle is extremely complex, and a large portion of our work went to streamlining Isabelle components to enable step (ii) above, notably the Isabelle PIDE infrastructure for incremental processing of proof documents. Thirdly, the resulting exports of the Isabelle libraries were significantly larger than any exports we had handled previously. Therefore, we had to develop new optimizations both on the Isabelle and on the MMT side to be able to carry out step (iii) above.

Repeating such an advanced MMT integration for other proof assistants must revisit the particular technology found there. In particular, proof assistants can vary widely in how the building of large projects and of dependencies between projects are handled. For example, Coq uses a decentralized library with hundreds of repositories and consequently uses sophisticated tools for repository management and continuous integration, e.g., the piCoq tool [42] to manage build processes in a fine-grained manner. Thus, the corresponding problem is more complex for Coq as it is for Isabelle, where the library is more centralized and the build management is tightly integrated with the kernel. piCoq already involves some Java-based components, which might help integrate it with the MMT Scala API.

Contribution and Overview. We apply our approach to Isabelle [44]: we present a definition of the Isabelle logical framework in MMT and an export feature for Isabelle logics and libraries. We exemplify the latter by exporting the standard Isabelle distribution [19] and the *Archive of Formal Proofs* [1]. The translated libraries are available at <https://gl.mathhub.info/Isabelle> as compressed OMDoc files.

We present preliminaries about Isabelle and PIDE as well as OMDoc and MMT in Sections 2 and 3. Then we describe the logical and the technical aspects of the export in Sections 4 and 5. We sketch some applications enabled by the export in Section 6.

It is difficult to estimate the total workload covered by this paper because it builds on decades of implementation work in both Isabelle and MMT, much of which was never published in itself. But concretely for this particular export, we spent about 1 person-month on the overall design of the translation and the implementation, 6 person-months on the implementation on the Isabelle side, 1 on the MMT side, and 1 on administrative parts and dissemination of the results.

2 Isabelle and PIDE

The Isabelle Platform. Isabelle [43, 44] is a generic platform for formal logic tools. Its foundation is the *Pure logical framework* by Paulson [43] based on a minimal intuitionistic higher-order logic with declarative natural deduction proofs. Isabelle/Pure is used to represent object-logics like Isabelle/FOL, Isabelle/ZF, and the most widely used Isabelle/HOL based on Church's simple type theory and Gordon's HOL [11].

Extra-logical tools are implemented in the *Meta Language (ML)* in LCF style [12]. Isabelle/ML has full access to the symbolic representation of the logic and provides many add-ons such as concrete syntax and context management for proof tools. The ML compiler and toplevel environment are managed within the same formal context as the logic, so ML declarations follow the structure of theory specifications and proofs.

ML is mainly used for pure mathematical programming with limited access to the physical world. Additionally, Scala (running on the Java platform) is used for external tooling: it manages ML processes, formal sources, and the resulting content, and provides an outer shell for Isabelle systems programming with access to GUI frameworks, TCP servers, database engines, etc. The programming style of Isabelle/Scala resembles Isabelle/ML, and some important modules are available on both sides (e.g. formatting of pretty-printed text).

Isabelle’s Prover IDE framework PIDE [49] integrates all development into the semantic text editor Isabelle/jEdit [52]. While the user is composing text, PIDE provides real-time markup about its meaning – rendered as, e.g., text color, squiggly underline, tooltips, hyperlinks, icons in the border. The Prover IDE supports ML development as well: users can edit theory sources with embedded ML modules directly, while the ML compiler does static checking and dynamic evaluation on the spot. Thus Isabelle has no need for externally compiled modules, in contrast to, e.g., Coq plugins.

More recently, Isabelle/PIDE has been refined to support *headless mode*, which lets a function in Isabelle/Scala observe this markup while a formal library is processed in Isabelle/ML. Compared to traditional batch-builds, headless PIDE provides more detailed feedback from the prover and more flexibility in dynamic loading and unloading of theories. In particular, it allows the processing of Isabelle content for other purposes than editing it in a GUI. This is the central interface that we use in the work reported in this article.

Isabelle Libraries. The standard distribution of Isabelle includes the Isabelle/HOL library with many examples, but the bulk of applications is in the *Archive of Formal Proofs (AFP)*, which is organized like a scientific online journal. In April 2020, AFP had 528 articles by 347 authors, comprising a total of 130 MB of source text in 5343 theory files.

Formal processing of the Isabelle distribution plus AFP requires \approx 46h CPU time or 13h elapsed time, using standard hardware with 8 CPU cores and 16 GB RAM. Such `isabelle build jobs` [53] produce heap images for the internal state of Isabelle/ML and optional HTML/PDF documents that resemble conventional mathematical texts.

Library Structure. Isabelle libraries consist of formal documents [50] structured according to session definitions, theory imports, and commands within theories:

- A *session* is a collection of theories with optional \LaTeX document preparation. It may refer to a single *parent session* and multiple *import sessions* (to reuse some of their theories by reloading their sources within the original session name space). For example, the session `HOL` is the basis for most applications, and the session `HOL-Analysis` is a substantial library of standard mathematics. In the AFP, each entry (or “article”) usually corresponds to a single session with its own setup for the published PDF document.
- A *theory* is a linear arrangement of commands corresponding to definition–statement–proof in conventional mathematical texts. The theory header imports multiple parent theories, taking a strictly monotonic *merge* of existing theories as basis for the new one. For example, theories like `HOL.Nat`, `HOL.List` are stepping stones towards `Main` and `Complex_Main`, which have global names and are the key entry-points for applications.

- A *command* is a functional update on the theory context (or proof state) using concrete syntax within the source file. Command syntax may embed user-defined sublanguages delimited as so-called “cartouches”, e.g. **ML** $\langle val a = 1 \rangle$. Theories may define new commands at any time – even Isabelle/Pure itself is defined in user-space relying only on the **ML** command for bootstrapping. For example, the commands **definition**, **inductive**, **fun** define constants and automatically prove characteristic theorems over them, while **lemma**, **proof**, **qed**, **by** are for proofs written in the Isar proof language.

The overall graph of sessions and theories is managed by Isabelle to exploit parallel processing within multithreaded ML (and Scala). For example, a theory could already be finished on the surface but some of its proofs still pending in parallel forks. Isabelle/Scala provides operations to explore sources down to command spans (keyword with argument tokens), without requiring a prover process to interpret them in the formal context.

Library Processing. The library sources are processed by feeding them to the Isabelle/ML session managed by Isabelle/Scala. This constructs formal meaning that is a-priori opaque, i.e., a matter of the private context of the logic or user-defined sublanguage. In order to expose some aspects of the meaning, Isabelle/ML supports several formal message channels:

- *Output* of regular messages, warnings, errors, etc. with text that typically refers to logical types and terms. Pretty-printing with blocks and breaks is supported by default: the front-end usually does the formatting based on precise window and font sizes. For example, the command **term** turns its source argument into an internal term and pretty-prints the result with markup to link constants to their definitions.
- *Reports* to assign markup to existing input sources (with precise positions). For example, after reading a term from the source text its precise positions of free and bound variables are reported as XML markup elements `<free/>` and `<bound/>`. The editor turns this into the usual Isabelle color scheme of blue vs. green variables.
- *Exports* to attach arbitrary blobs to a theory (with hierarchic names separated by slash). For example, the command **export_code** turns Isabelle/HOL specifications into program source (for SML, OCaml, Scala), and the result becomes an export artifact of the enclosing theory. Thus the current version of input sources (e.g., an open buffer in Isabelle/jEdit) is augmented by the result of **export_code** seen as a mathematical function; the editor shows the result via the *virtual file-system* URL `isabelle-export:` within its File Browser, independently of the accidental state of the physical file-system.

The exposed aspects of document meaning are stored within the *session database*. For conventional batch-builds, that is an SQLite database file used like an archive with XZ-compressed entries, and the command-line tool `isabelle export` lists and extracts its content. For PIDE processing, the database consists of Scala values within the document snapshot and may be explored via user-provided Scala functions, e.g., for GUI painting of annotated document source. It is also possible to write out the data to another database (e.g., PostgreSQL is supported routinely), or in a completely different application, which is what we do in the OAF-style export reported on in this article.

To support the latter, Wenzel has modified the processing to allow for application-specific ML functions for presentation. Whenever a theory node with all its imports is fully consolidated (parallel proofs finished), user-defined ML functions can access its list of commands paired with the internal theory context at each step.

Isabelle/Pure and Isabelle/HOL provide standard presentation functions to expose core material from the logical context, guarded by option `export_theory`. Results are exported to the session database, using a private XML representation, Isabelle YXML transfer syntax, and

XZ compression of the resulting blob. This works both for batch sessions (`isabelle build`) and for headless PIDE sessions (`isabelle dump`). Thus, with the current infrastructure, the request by Kohlhase from 1999 could be fulfilled on the spot via `isabelle dump -B ZF`, but instead of digesting raw XML/YXML data it is better to use typed APIs in Isabelle/Scala (by using module `Export_Theory` as we do in Section 5.1).

3 OMDoc and MMT

Language. OMDOC [27] (short for **O**pen **M**athematical **D**ocuments) is a semantics-oriented XML-based markup format for STEM-related documents. It conceptualizes mathematical objects in three levels as seen in Table 1: the *object* level for mathematical formulas and their presentations, the *statement* level for definitions, theorems, proofs, etc, and the *theory* level for collections of statements. Each level comes in two dimensions for the formal representations of the content addressed to mathematical software systems and the narrative structure addressed to humans. Higher levels may contain expressions of lower ones, and mixtures of dimensions are allowed, leading to a overall format that can handle flexible levels of formality (see [31] for a discussion).

■ **Table 1** Three level & two dimensions in OMDoc.

level	formal	narrative
object	OpenMath	presentation MathML
statement	sequents	paragraphs + cues
theory	theories/views	sections, etc.

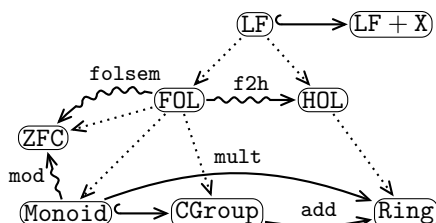
Even at the early state in 1999, OMDOC already had this general architecture and was therefore well-suited in principle for representing Isabelle content, in particular the Isar proof language [54] that was new at the time. But the formal part of OMDOC was purely descriptive and lacked a rigorous semantics. In particular, the role of the logical systems needed for formally stating mathematical properties was almost fully unspecified beyond the idea – inherited from OpenMath – that logics are theories as well.

Later MMT (Meta Meta Theories) [48] re-conceptualized and refined the formal fragment of OMDOC, greatly enhancing both rigor and expressivity. It models formal objects and statements using logical frameworks, in particular the judgments-as-types paradigm, and bases OMDOC’s theory level on the category of theories and theory morphisms following the development graphs approach [2]. The former allows for fine-grained specifications of the semantics of individual objects, and the latter allows for inducing and translating knowledge across theories. A new *meta-theory* relation links a logical framework to the logics defined in it, thus formalizing the “logics-as-theories” approach.

The MMT System. The OMDOC/MMT language is implemented in the MMT system (Meta Meta Toolset; see [47]), which provides an API for the language constructs at all levels and provides both logical services such as type reconstruction and rewriting and knowledge management services such as IDE and HTML presentation and browsing of libraries.

Because it avoids committing to a specific semantics or logical foundation, foundation-dependent services and features (e.g., type reconstruction) are implemented by splitting the algorithms into a foundation-independent kernel that is user-extensible with foundation-specific rules. For example, the logical framework LF [15] is implemented using about 10 rules taking only a few lines of code each.

Theory Graphs. Theory graphs are diagrams in the categories of theories and morphisms. The possible morphisms in MMT are **inclusions**, which import all declarations from the domain to the co-domain, **structures**, which are like includes but copy and translate all declarations, **views**, which are semantics-preserving translations from domain to codomain, and the **meta-theory**-relation, which behaves like an include for most purposes.



■ **Figure 1** Meta-Levels in OMDoc/MMT.

Figure 1 shows an example of a typical setup of formalizations in MMT: Dotted lines represent the meta-theory-relation, hooked arrows are includes, squiggly arrows represent views, and the normal arrows represent named structures. Here LF is used as a logical framework to define some logics, which are then used as meta-theories for algebraic theories. We see three pragmatic levels: the logical frameworks at the top, logics in the middle, and the domain theories at the bottom. Meaning trickles down from the theories at the top (the ones without meta-theories), which are implemented directly in MMT/Scala as described for LF above. This setup can even encode model theory theory morphisms into semantic theories like ZFC set theory.

4 Logical Aspects of the Translation

The logical basis of our export is a definition of Pure in the MMT system. MMT allows defining a wide variety of logical frameworks, and we use PLF as a starting point, a polymorphic variant of LF [15] that already exists in the MMT standard library [37].

4.1 Type System and Logic

Types, Terms, Propositions. We use a shallow embedding of Pure in PLF. Besides simplicity, this has a critical scalability advantage: a deep embedding would lead to substantially larger PLF-expressions when already our shallow embedding ended up yielding the largest export size we had ever attempted (since then eclipsed only by our analogous export for Coq [38]). Consequently, as Pure uses shallow polymorphism (type variables bound at the outside of declarations), we cannot use LF itself but need to extend it with shallow polymorphism. That is why we use PLF instead.

Using a shallow embedding, most Pure primitives are represented as their PLF-counterparts: Pure-types and terms are represented as PLF-types and terms. This includes in particular Pure’s simple **function** types, λ -abstractions, and application.

The remaining primitives can simply be declared as PLF-constants. That yields a PLF-theory containing in particular the constants

- `prop` : type for the type of **propositions**,
- `ded` : `prop` \rightarrow type mapping each proposition φ to the type `ded` φ of **proofs** of φ .

That is the bare minimum to connect Isabelle/Pure to PLF: the remaining connectives are produced from the regular export of the Pure theory itself, yielding further constants:

- $Pure.eq : \Pi_{a:type} a \rightarrow a \rightarrow \mathbf{prop}$ polymorphic **equality** (with implicit $\alpha\beta\eta$ -conversion),
- $Pure.all : \Pi_{a:type} (a \rightarrow \mathbf{prop}) \rightarrow \mathbf{prop}$ for the polymorphic binder for **local parameters**,
- $Pure.imp : \mathbf{prop} \rightarrow \mathbf{prop} \rightarrow \mathbf{prop}$ for the constructor for **logical entailment**.

Relative to these declarations, it is straightforward to translate all Isabelle types, terms, and propositions.

Proof Terms. Like LF but unlike Pure, PLF offers dependent types. These are not needed for representing the simply-typed Pure language but are helpful to concisely represent Pure-proofs as PLF-terms in Curry-Howard style. Thus, Pure proof terms can be exported analogously to types, terms, and propositions. However, in practice, we only export proof terms for small examples because proof terms for actual Isabelle/HOL are far too big. After our work on Isabelle, we conducted a similar export for Coq in [38]. Here we included proof terms, and the sizes, while large, remained manageable. But due to the lack of Coq-style implicit computation, we expect Pure proof terms to be even larger.

However, there is a separate, deeper reason to defer proof exports: it is still unclear what the best way to export proofs is. The export of low-level proof terms is straightforward, but the proof objects are huge and have only limited value (independent proof checking being the main one). The high-level proofs seen by the user are much more interesting but lack the information inferred by the prover.

Therefore, we opted for exporting all proofs as dummy terms that carry only the information that the theorem was checked by Isabelle and which dependencies were used. Additionally, we include, as an informal narrative text, the command-source of the Isar text: this treats the whole proof as one unit, without the hierarchical structure of Isar proofs (see also the discussion in 4.4 and 6.5 below).

4.2 Declarations

Foundational Declarations. It is straightforward to represent the foundational declarations of Pure theories as PLF-declarations as follows:

- **Pure-type operators** a of arity n as n -ary PLF-constants

$$a : \mathbf{type} \rightarrow \dots \rightarrow \mathbf{type} \rightarrow \mathbf{type}$$

- **Polymorphic Pure-terms** c of type A using type variables a_1, \dots, a_n as PLF-constants

$$c : \Pi_{a_1:type} \dots \Pi_{a_n:type} A$$

- **Polymorphic Pure-axioms** s with type parameters a_1, \dots, a_n asserting proposition F as PLF-constants

$$s : \Pi_{a_1:type} \dots \Pi_{a_n:type} \mathbf{ded} F$$

All three kinds of declarations may carry definitions, which can be represented by giving the PLF-constant a definiens. This is used only for type operators and term *abbreviations*. HOL type definitions are a special case of high-level declarations as described below, and Pure term definitions are mapped to definition-less constants with defining axioms (multiple ones in case of overloading). Additionally, theorems are represented using the proof as the definiens (as described above).

Identifiers. Isabelle assigns to each foundational declaration a unique identifier. It uses separate namespaces for types, terms, and theorems and usually qualifies their names by the base name of the enclosing theory. Every theory exists within an Isabelle session, whose name usually qualifies the theory’s base name. Both qualification schemes are optional – there is no strict enforcement.

For reusability, it is preferable to use a single namespace (to ensure globally unique identifiers for all declarations) and to use a uniform naming schema for all identifiers. Moreover, MMT requires all names to be globally unique by qualifying them with an ownership-defining URI. So we have chosen the following naming scheme for all declarations:

`https://isabelle.in.tum.de?long-theory-name?entity-name|entity-kind`

where *long-theory-name* is the session-qualified theory name, *entity-name* the declaration name within the theory context, and *entity-kind* its name space: notably `type`, `const`, `thm`, or other name spaces of user-defined concepts. For example,

`https://isabelle.in.tum.de?HOL.Nat?Nat.nat|type`

refers to the type `nat` of natural numbers in the theory `Nat` in the session `HOL` of the main Isabelle library. The seemingly redundant repetition of `Nat` is needed to cover corner cases, including some unqualified names in Isabelle/Pure.

High-Level Declarations. Isabelle provides a user-extensible set of high-level specification elements, whose semantics is defined by elaboration into foundational ones. Examples include HOL-type definitions or the definition of inductive data types and recursive functions. Similarly, the high-level specification contexts of locales and type-classes (see below) are elaborated into primitive concepts of the logic. Both are already covered by exporting their elaboration, but that results in representations without the high-level structure seen by users.

MMT provides a similar extensible declaration pattern mechanism [16, 39] so that we can use them to represent Isabelle’s high-level declarations in a structure-preserving way. We have so far carried out this effort only for locales and leave other elements to future work: it could be done by a generic Isabelle/ML interface for such specification elements such that the export works uniformly for all its instances. Then a manageable separate implementation effort would be needed for each specification element. However, because the individual specification elements were implemented by different authors and can be very complex, no single person could retrofit them to implement this interface, and a long-term community effort is required.

4.3 Module System

Theories. The MMT module system subsumes the expressivity of Isabelle theories and is available for every language defined in MMT such as PLF. Thus, all Isabelle theories (including those for logics like HOL) are represented straightforwardly as PLF-theories.

Locales. As the Isabelle logical framework lacks primitive support for “little theories”, a locale definition is elaborated into a constant definition (predicate) for the logical specification, together with extra-logical management of the resulting context and conclusions produced within it [25]; similar techniques are used for Isabelle type classes [13] on top of locales.

1:10 Making Isabelle Content Accessible in Knowledge Representation Formats

Without any special care, the export of locales merely shows these predicate definitions with theorems depending on additional parameters and premises. But this low-level elaboration is not what Isabelle users expect. Instead we refer to exported information about the original structure of locale specifications and map that to first-class theories in MMT. Subsequently, we illustrate this approach by a representative example.

Semigroups. Consider the following locale for semigroups. It declares (fixes) the binary operation (where we write $x*y$ for $op\ x\ y$), assumes the associativity axiom, defines the squaring function, and states a simple theorem:

```

locale sg =
  fixes op :: 'a → 'a → 'a (infixl * 70)
  assumes assoc: ∀ x y z. (x * y) * z = x * (y * z)
begin
  definition sq :: 'a → 'a where sq x = x * x
  theorem sqsq: sq (sq x) = x * sq x * x <proof>
end

```

Note that the universe of the semigroup is not declared explicitly. Instead, Isabelle locales treat any type variable that remains uninstantiated after type-checking as a type fixed in the locale. In our PLF representation, this convention is made explicit by declaring the universe a as a type and then treating all fixed types and operations uniformly. In the sequel, we use the words *structure* to refer to a tuple of values interpreting the fixed types and operations, and *instance* for a structure that satisfies the assumed axioms.

Translation by Elaboration. The locale's elaboration is represented as the following set of PLF-constants (where we again write $x * y$ for $op\ x\ y$ but op is now always a bound variable):

- one membership predicate that ranges over structures and a defining axiom for it that makes it true for instances:

$$sg : \Pi_{a:\text{type}} \Pi_{op:a \rightarrow a \rightarrow a} \text{prop}$$

$$sg_def : \Pi_{a:\text{type}} \Pi_{op:a \rightarrow a \rightarrow a} \text{ded } (sg\ a\ op) \Leftrightarrow \forall x, y, z. (x * y) * z = x * (y * z)$$

- for every definition, a global constant and a defining axiom for it, both abstracting over structures:

$$sg.sq : \Pi_{a:\text{type}} \Pi_{op:a \rightarrow a \rightarrow a} a \rightarrow a$$

$$sg.sq_def : \Pi_{a:\text{type}} \Pi_{op:a \rightarrow a \rightarrow a} \text{ded } d = \lambda_{x:a} x * x$$

- for every theorem, a global theorem abstracting over structures and relativized to instances:

$$sg.sqsq : \Pi_{a:\text{type}} \Pi_{op:a \rightarrow a \rightarrow a} \text{ded } (sg\ a\ op) \Rightarrow \forall x. SQ(SQ\ x) = x * (SQ\ x) * x \\ := \text{(proof omitted)}$$

(abbreviating $sg.sq\ a\ op$ as SQ).

Note that Isabelle's elaboration introduces the function $sg.sq$ for all structures even though it is only defined for instances. This is sound in the special case of Isabelle because function types are simple and all types are non-empty (which makes adding unspecified operations conservative) and because all locale theorems are relativized to instances.

Reconstruction of Isabelle Locales s MMT Theories. By elaborating locales into global declarations, some information about the modular structure is lost. To allow for preserving that structure, we additionally and redundantly export every locale as a PLF-theory with the following local declarations:

- a primitive constant for all fixed types and operations and assumed axioms:

$$a : \text{type}$$

$$op : a \rightarrow a \rightarrow a$$

$$assoc : \text{ded } \forall x, y, z. (x * y) * z = x * (y * z)$$

(writing $x * y$ for $op\ x\ y$),

- a defined constant for each definition and theorem:

$$sq : a \rightarrow a := \lambda_{x:a} x * x$$

$$sqsq : \text{ded } \forall x. sq(sq\ x) = x * (sq\ x) * x := [\text{proof omitted}]$$

This nicely conforms to the intention of Isabelle locales as extra-logical add-ons to the Pure logic. We represent sublocale relations and locale interpretations as PLF theory morphisms accordingly (by re-using exported information from Isabelle locale management).

Type Classes. Type classes are a special case of locales with some add-on infrastructure, notably for type inference. A locale may become a type class if it has exactly one free type variable 'a.

If sq is instead declared as a type class, the following additional declarations are present:

- for every fixed operation, a global constant abstracting only over the single fixed type:

$$sg_class.op : \Pi_{a:\text{type}} a \rightarrow a \rightarrow a$$

- for every assumed axiom, a corresponding global axiom relativized by the membership predicate sg of the locale (instantiating the fixed operation op with $sg_class.op\ a$):

$$sg_class.assoc : \Pi_{a:\text{type}} \text{ded } sg\ a\ (sg_class.op\ a) \Rightarrow \forall x, y, z. (x * y) * z = x * (y * z)$$

(writing $x * y$ for $sg_class.op\ a\ x\ y$)

- for every definition, a corresponding global constant with a defining axiom,
- for every theorem, a corresponding global theorem.

4.4 Ontology

The description above covers the translation of all logical content. But it is useful to additionally export a high-level abstraction of the library ontology in semantic web style. This includes all named entities (locales, theorems, etc.) and their interrelations but excludes all complex objects (types, terms, proofs).

Such an ontology export is easier to maintain efficiently, e.g., using RDF triple stores. And it is sufficient for many important applications such as querying the dependency relation between declarations. Additionally, it can easily include metadata such as check times.

Isabelle/MMT performs such an RDF/XML export as well, see also 5.3 for the amount of relational information. We originally presented this RDF export in [9] together with an Upper Library Ontology (ULO) that describes and provides a uniform vocabulary of classes and relations for all proof assistants; therefore, we mention only a few recent improvements

here. The relational ontology also captures some aspects of inductive and primitive recursive definitions (via the binary relation `ulo:inductive-on`). Most importantly, our export now fully covers dependencies, spanning a large dependency graph over the source text: it relates via the binary relation `ulo:uses` every theorem statement with every used constant and every proofs with every used theorem.

5 Technical Aspects of the Translation

The majority of the export is not OMDOC-specific and carried out on the Isabelle side; this appeared first in the official release Isabelle2019 (June 2019), but the present paper uses the reworked and simplified version of Isabelle2020 (April 2020). Being integrated into Isabelle has the advantage that most of our work can be immediately reused for exports into other formats than OMDOC. Only little OMDOC-specific code is necessary for building and serializing the XML objects in OMDOC format. For this, we use the MMT API for OMDOC, which is also written in Scala and therefore directly callable from PIDE. This code is now part of the MMT distribution (first in release 14 from November 2018).

The resulting inter-dependency between the code bases is handled as follows: if the MMT directory is registered to Isabelle as *component*, it provides a tool `isabelle mmt_build` (shell script) to build MMT with Isabelle support enabled. The resulting `mmt.jar` will provide further tools `isabelle mmt_import` and `isabelle mmt_server` (in Scala) to perform the import and view its results. Users merely need to invoke, e.g., `isabelle mmt_import -B ZF`.

5.1 Export from Isabelle

Isabelle/Scala provides a standard module `Export_Theory` to expose theory content to other tools via a statically typed API that imitates Isabelle/ML datatypes for types and terms. The communication between Isabelle/ML and Isabelle/Scala works via untyped XML trees, without any special tricks about meta-programming. Instead, sources in both languages reside next to each other in the official Isabelle repository, are manually updated accordingly.

A first version of the Isabelle export facility appeared in Isabelle2018 (August 2018). It was originally motivated by early versions of Isabelle/MMT, and has grown into an independent Isabelle service. It is supported by command-line tools like `isabelle export` and `isabelle dump` [53]; `isabelle build` with option `export_theory` exposes logical content as follows.

- Foundational theory content of the Isabelle/Pure *logical framework*: **types** (base types and type constructors), term **constants** (including functions, binders, quantifiers as higher-order constants), **axioms** (including equational axioms that count as primitive definitions), and **theorems** (propositions with a proof). Actual proofs are not exported by default – they are prohibitively large. The option `export_standard_proofs` provides proof terms in a standardized format that facilitates import in other tools, but this only works for small examples so far.
- Constant definitions of Isabelle/Pure, as a relation between a single constant with multiple axioms. Overloading in Isabelle means that a polymorphic entity is characterized on multiple (non-overlapping) type instances. The majority of constants are non-overloaded, with exactly one equational axiom to express its definition. This relation of constants to their defining axioms is exported, too.
- Type definitions of Isabelle/HOL in the sense of Gordon and Pitts [45]. This axiomatization scheme can be interpreted definitionally within the standard semantics of the HOL logic. Isabelle/HOL provides a separate module to create new types via that mechanism. Some key information is exported: the old representing type, the new abstract type, the name of the morphisms between the two with the axiom stating the relation.

This allows recovering HOL typedefs faithfully, where Pure theory content would only show the individual particles. It also serves as an example to “query” derived specification mechanisms in Isabelle/ML, to expose its own level of abstraction to the exporter.

- Term constants with indication of derived specifications mechanisms, e.g. **primrec** functions, **inductive** or **coinductive** relations. This works by querying generic information in Isabelle/Pure about functional or relation specifications (also known as “Spec Rules”). The Isabelle/HOL implementations provide this data on their own account. This merely provides a rough classification of term constants at a very abstract level. The full complexity of Isabelle/HOL specification mechanisms is more difficult to capture: it would mean to follow many implementation details, including ones that have changed fundamentally over the years of ongoing Isabelle development.
- Dependencies of proven theorems wrt. types, consts, theorems, as recorded by the Isabelle inference kernel: This spans a large dependency graph over the document in terms of the primitive logic – extra-logical aspects are missing (e.g., dependency on notation). Partial support for these *proof constants* had been part of the Isabelle codebase over many years, but we had to rework this substantially to make it suitable for our application.
- Locales in the sense of Ballarin [3] and type classes as special locale interpretations in the sense of Haftmann and Wenzel [13, 14]: The export of locales preserves some of its internal structure, notably the locale dependency relation stemming from the construction of locales and sub-locales (by definition), as well as later locale interpretations (by proof). These are then exported as MMT theory morphisms. For type classes, the export shows the canonical locale interpretation but without an explicit connection to the type class. This would have to be a type-indexed family of MMT theory morphisms.
- The order-sorted algebra of *type classes* (subclass relation) and *type arities* (image behavior of type constructors wrt. type class domains and ranges) in the sense of [40]: This allows reconstructing Isabelle’s built-in type class reasoning by an external program (for example, an application could give it to a separate process running Isabelle/Pure and reuse the original implementation in module `Sorts Isabelle/ML`). An alternative is to imitate these operations in a different programming language.²

Formal entities have two name components: *kind* (to distinguish the namespace) and *full name* (usually with the theory base name as qualifier). In addition, there is an *external name* for printing (partially qualified according to standard namespace policies), a *source position*, and a *command span identifier*. The latter allows in particular arranging the content according to the order in which it occurs in the source text so that exported types, constants, theorems appear as a digest for each specification element in the text (e.g. for **definition**).

Moreover, if the target format of the export supports references to the original source, this can be used to attach such a reference or even the entire source fragment to each formal entity. We do that for our OMDOC export.

5.2 Import into MMT

The entities listed in Section 5.1 can be serialized almost directly as MMT constants relative to the PLF framework as described in Section 4. That is not surprising as much of that work motivated by the present export in the first place. Figure 2 shows the MMT browser displaying an example that is very small and thus includes proof terms. Note how every formal declaration is preceded with an informal narrative fragment containing the original source text, this is for the orientation for Isabelle users.

² Isabelle/Scala does not provide any type-class reasoning on its own, because it is meant to be for external system management only. Logical operations are done properly in Isabelle/ML.

In the sequel, we describe a few specific adaptations of the term language that were required to reconcile traditional Isabelle/ML representations with the more conventional λ -calculus of PLF in MMT.

Type arguments for constants. The traditional representation of polymorphic constants in Isabelle and the HOL family [45] is to give the full *type instance* at each occurrence in a term, instead of the *type arguments* that produce the instantiation of the general type schema. For example, constant `id :: 'a => 'a` occurs in particular terms as the pair $(\text{id}, \tau \Rightarrow \tau)$ for the respective type τ . This is both redundant (because the type instances are usually bigger than the type arguments) and inconvenient (because it is more difficult to obtain the type arguments from the instantiations than the other way around). In contrast, PLF treats `id` as a function with dependent type $\Pi_{a:\text{type}} a \rightarrow a$ and occurrences are just applications $(\text{id } \tau)$.

Isabelle/ML provides operations to switch between the two representations within a given context of constant declarations. Our theory export always uses the second form with type arguments: this reduces the size of exported material and allows importing terms into PLF without again referring to the environment of constant declarations.

Variable names. Isabelle variables come in various flavors: free variables (e.g., `x`), schematic variables with index (e.g., `?x10`), and bound variables (e.g., `x` in $\lambda x:\tau. x$) which is notation for the de-Bruijn index abstraction `Abs (x, τ , B.0)` where `x` is retained as a comment).

To fit smoothly into the λ -calculus of PLF, schematic variables are renamed to fresh free variables. Since schematic variables are morally like a universal quantifier prefix, this preserves the logical meaning of a statement. And bound variable comments in abstractions are renamed locally to avoid clashes with free variables in the same scope. Thus the `Abs` comment can be used literally in PLF as a named abstraction ignoring the unnamed de-Bruijn index representation of Isabelle.

Type class constraints. Isabelle type variables are decorated with type class constraints, e.g., `'a::order` for types that belong to the class `order` defined in the Isabelle/HOL library (e.g., `nat` with its standard order): this links certain operations to overloaded term constants (e.g., `less :: 'a => 'a => bool`) and ensures logical premises on these operations (e.g., stating that `less` is a strict order on the type).

Isabelle type class operations are managed by extra-logical means to eliminate the implicit overloading. In PLF this merely results in multiple constant definitions for different type arguments. Class premises become logical constraints in a straight-forward manner: a type class is a predicate over types in PLF. So `'a::c` means that the predicate `c` applied to type `'a` holds. Statements with class constraints $\varphi('a::c)$ are augmented by a prefix of preconditions `'a::c \implies $\varphi('a)$` , effectively eliminating the constraint within the logic.

5.3 Statistics for Isabelle/AFP

Our test hardware for the MMT export of Isabelle/AFP is a server machine with 40 CPU cores (80 hardware threads), 128 GB RAM (2 NUMA nodes), and fast SSD storage. Below, we give an overview of the material for Isabelle2020 (April 2020) with MMT/52adb5e338811e [20] and AFP/91f1cdbefc0 [1]: These sources consist of 680 sessions distributed over 7,027 files comprising 160 MB of theory text (30 MB XZ-compressed). The exported content comprises

- 7,027 theories and 5,291 locales (“little theories”), including 1,236 type classes,
- 2,116,638 individuals (11,724 `type`, 204,404 `const`, 236,186 `axiom`, 1,497,689 `thm`).



Figure 2 Disjunction in Higher-Order Logic: definitions, theorems and proof terms.

1:16 Making Isabelle Content Accessible in Knowledge Representation Formats

- 400,996,957 relations, including 386,325,246 `ulo:uses` (i.e. the overall dependency graph of `type`, `const` and `thm` items)
- 65 GB OMDoc/XML (310 MB XZ-compressed)³

The entire process of Isabelle/PIDE document checking, export to MMT, and serialization as XZ-compressed XML requires 80 GB RAM, 8 CPU cores, and 22h30 elapsed time. Thus, compared to an elementary batch-build, our export requires around 2 times the memory and 2–5 times the elapsed time (mainly because Isabelle/MMT uses less parallelization than `isabelle build`). We emphasize that these resource figures are for the *entire* AFP, including the special sessions tagged as `slow` or `large` which are often omitted because they take a lot of resources to process.

The size of the exported OMDoc data structures is linear in the size of the original sources, increased by about factor 10 in XZ-compressed form. This increase in size is a gain, not a deficiency – it stems from the fact that the exported XML contains substantial additional information that is implicit in the sources but extremely difficult to infer: all occurrences of symbols are disambiguated and exported with their unique URIs; the exported XML elements carry source references, i.e., URIs that link to the corresponding location in the source; all type arguments of occurrences of polymorphic constants and all types of bound variables are included in the XML even if omitted in the sources; and all theorems automatically generated by Isabelle are included in the export. We could suppress some of this information, but that would defeat the purpose of our export: only Isabelle can infer all details, and handing it to other tools is our export’s main value. The uncompressed XML files are much larger because they are very verbose and optimized for context-free processing. But we never write the XML directly to the file-system: all reading and writing of XML is filtered through XZ compression.

5.4 Maintainability

When developing proof assistant library exports, the challenge of maintainability is often overlooked or underestimated. This is partly caused by the incentives of the academic system that rewards quickly published results rather than long-term sustainable ones. We have consciously taken several steps to ensure maintainability.

Firstly, we use statically-typed Scala APIs as much as possible, both in the export from Isabelle and in the import into MMT. Almost all the new code we wrote for the occasion was immediately integrated with the existing abstract interfaces. The remaining glue code that connects Isabelle’s abstract export with MMT’s abstract import comprises only a few thousand straightforward lines of code.

Secondly, wherever possible we wrote new code in the Isabelle repository rather than the MMT repository. This forces future Isabelle development to maintain our abstract code, in particular when PIDE data structures change. Concretely, we pushed only the parts of the code that actually depend on the MMT data structures to the MMT repository. That portion consists of only about 2000 lines of code, mostly straightforward code for creating instances of the MMT data structures. The rest of the export code is generally reusable for other Isabelle exports and pushed to the Isabelle repository and already released as an official Isabelle feature. In fact, this design has already proved beneficial as Wenzel was able to reuse the Isabelle part of our code in a recent export to Dedukti (still unpublished).

³ <https://gl.mathhub.info/Isabelle/Distribution/commit/db1009a326c8> and <https://gl.mathhub.info/Isabelle/AFP/commit/346f28873c9f>

Finally, the fact that Isabelle and MMT can communicate via the Java VM has proved a huge advantage for maintainability. We were able to design the code in such a way that MMT is an optional plugin component for Isabelle and vice versa. Thus, users running Isabelle can simply register MMT as a plugin with Isabelle and then run `isabelle mmt_import` on the command-line.

Whenever a new Isabelle release is published, it will be a matter to update some statically-typed Scala functions for Isabelle/MMT. Informed by our experience of multiple similar exports, we judge this one to be the most maintainable export of a proof assistant library so far, in fact by a wide margin.

6 Enabled Applications

Our work now allows exporting entire Isabelle libraries into a format that can be easily read by third-party applications in a robustly maintainable way. A major motivation for this work was enabling applications that use this exported data. However, it remains open which applications should be better realized directly in Isabelle and which should be based on MMT. Critically, our export abstracts from most idiosyncrasies of Isabelle’s logic, implementation, and library structure. That has advantages and disadvantages.

On the positive side, any application that does not significantly depend on Isabelle’s code base (e.g., search or dependency management) or explicitly rejects using it (e.g., representations in a logical framework or external proof checking) benefits from the uniform representation in the relatively simple language of MMT. On the negative side, any application that should be tightly integrated with Isabelle may be better realized natively in Isabelle. This includes in particular applications that offer proof advice or rewrites/generates Isabelle data structures or Isabelle sources.

In some cases combined approaches may be indicated such as a small native addition to Isabelle that connects to a service implemented on top of the MMT representation (and possibly running on a high-performance remote server). For example, search services could be realized well in this way. However, even when a native implementation that ignores the import into MMT is indicated, our work can provide substantial benefits. Any such native implementation will likely benefit from our streamlining and scaling up of Isabelle’s export capabilities that allow integrating such applications with Isabelle.

Ultimately, the assessment which of these effects dominate must be made on a case-by-case basis for every application. In the sequel, we sketch some applications enabled by our work where we expect the advantages to dominate.

6.1 Clarification of Isabelle/Pure in Terms of MMT/PLF

The Isabelle/Pure framework [43] is historically connected to Edinburgh LF, but it has its own distinctive style that can obscure important aspects. The documentation [51, §2] refers to related formulations of λHOL within the setting of Pure Type Systems (PTS) due to Barendregt and Geuvers [4] and gives informal explanations (in \LaTeX) about how to understand Isabelle-specific concepts like schematic variables or type-classes.

Instead of Isabelle folklore and informal explanations in the documentation, our translation to PLF within MMT elucidates many concepts of Pure more formally. In particular:

- The three levels of λ -calculus for function spaces (higher-order abstract syntax), universal binding of local parameters (quantification), logical entailment of rule statements (implication) become just one dependently-typed λ -calculus.

- Implicit polymorphism becomes explicit as abstraction and quantification over types.
- Up to scalability issues, proof terms – which are an optional add-on to the Pure logic – become plain λ -terms as definiens for theorems.
- Type class constraints become explicit as predicates applied to types. Concretely, there are two possible representations for extra-logical constraints: `'a::c` and intra-logical predication `OFCLASS('a, c_class)`. Both are turned into the obvious term `c a` for `c :: type => prop` in PLF).

Still lacking in our export is the explicit treatment of *type class parameters*: as in Isabelle/Pure, the PLF theory treats instance-specific definitions as a collection of axioms that are associated with a generically typed constant. A more sophisticated translation could try to make a dictionary construction, to turn type class parameters into explicit function parameters everywhere.

6.2 External Proof Checking

An often asked-for application of an Isabelle export is independent re-verification. It may appear straightforward to use our export as the input of a separate application that specializes on re-checking proofs. However, while this is certainly one of the intended uses, it would be naive to assume that our work is more than the first of multiple steps towards this goal. In the sequel, we describe the remaining two obstacles: scalability and adequacy. These obstacles are not inherent to our approach. We expect any future solution to external proof checking to build on our approach or to recreate something comparable.

Regarding **scalability**, it is indeed straightforward to write a proof-checker for the Pure logic underlying Isabelle. In fact, the MMT formalization of Pure induces a proof-checker for Isabelle out of the box. Similar framework-induced checkers can be built easily in implementations of LF-like frameworks such as Dedukti. Moreover, the complexity of these checkers would typically be linear in the size of the proofs and thus very feasible. It is even possible that checking the proofs could be faster than the file-system access needed to read the proofs in the first place.

But we do not expect such straightforward checkers to be able to handle the size of the proofs in the library: the size of individual proofs, if naively encoded, may very well exceed the memory capacity of typical checkers.⁴ Thus, additional investments are needed for handling large proofs, such as structure sharing, inferring omitted trivial steps, or streamed processing that can check a proof without loading it in its entirety. These technologies are known in principle, but applying them to Isabelle/AFP remains substantial future work.

Regarding **adequacy**, note that our export is foundational in the sense that it exports the representation relative to the Pure logic in Isabelle's kernel, which arises from the original user input through a series of highly non-trivial transformations (elaboration). Fully re-checking the proofs that result from elaboration is only one of two necessary conditions. The other one is *conservativity* of elaboration, i.e., the requirement that elaboration does not translate an unprovable statement to a provable one. Depending on how many advanced Isabelle features are used in a problem statement, trusting the conservativity of elaboration may be a bigger leap than trusting the correctness of the proofs.

But conservativity is extremely difficult to establish. The most direct way would be to specify the semantics of Isabelle's surface syntax and then prove Isabelle's elaboration algorithms correct relative to it. Given the complexity of elaboration, this remains out of reach in the foreseeable future.

⁴ Early experiments conducted with parts of the Main theory context of Isabelle/HOL produce hundreds of megabytes of proof terms in textual representation.

6.3 Dependency Management

The classic model of Isabelle/PIDE [49] document markup merely provides a record of formal entities that are *explicitly visible* in the source text. Due to some reworking of the inference kernel by Wenzel, there is now a detailed record of all `type` / `const` / `thm` entities that are *implicitly used*. This spans a rather large dependency graph over the original source: for Isabelle/AFP there are 400 million edges for 130 MB of theory text.

In the past, users have occasionally attempted to approximate this information for their own purposes, e.g. in the Levity tool [8], which exploits dependencies to move lemmas to adequate locations in the theory hierarchy.

Our ontological export (see Section 4.4) now includes a detailed record of both explicit source dependencies and implicit logical dependencies. With this information available in a standard format, more ambitious (and more robust) refactoring tools can be realized for Isabelle. Optionally, such refactoring tools can even be built in OMDOC/MMT to work uniformly for all systems that have exports similar to the one reported in this article.

6.4 Search

Because our export includes all logical information of the Isabelle content, it enables multiple search applications. For example, this would allow searching for expressions or names that are not explicitly part of the sources and only occur in inferred information. It also enables applying generic search systems to the Isabelle libraries.

As an example, we sketch a unification-based search service for the entire AFP based on MathWebSearch [33]. MathWebSearch maintains a substitution tree index that allows efficient unification queries over large collections of terms. Because it can index MMT terms, it can be directly applied to our export. Thus, users can explore the full background library without having it loaded into the prover process (which might require too much memory), or even without installing the prover at all (e.g., by using a web service for the AFP).

Concretely, the queries would be terms with free variables over some AFP theory, and the search results would be terms in the AFP that unify with the query. Because our export includes source references for all entities, these results can be linked to other resources (e.g., the location in the official AFP web site) or directly imported into PIDE.

The main remaining technical hurdle is the processing of the user's query. In order to match anything in the library, formal objects in the query must be processed and exported in the same way as the library. This includes the use of special forms for pattern matching, lists enumeration and comprehension etc. as well as type inference and type matching (with type classes). Moreover, the user must provide the right context in which to interpret the query.

An intermediate solution could run a prover session of reasonable size that contains the most relevant notation (e.g., `HOL-Analysis`) and process queries relative to it. These queries could then be exported and matched against the entire AFP.

We estimate that such a system is within reach of an ambitious Master's thesis.

6.5 Enabling Cross-Library Knowledge Management

Isabelle/MMT is one of multiple large exports of proof assistant libraries that we have conducted over the last few years. One of the original motivations of these efforts was to obtain multiple libraries in a uniform format in order to then develop cross-library and cross-prover knowledge management solutions.

These efforts are still at an experimental stage, and we only cite a few early results that could be extended to the Isabelle export:

- We have used alignments [21] to relate corresponding concepts in different libraries. These can be annotated manually or found by machine learning techniques [10]. Given a sufficient alignment coverage, we can then translate terms between libraries and use this to make systems interoperable.
- With the relational RDF/XML export of Section 4.4, we can use SPARQL queries using the Upper Library Ontology (see [9] for details) that return results from multiple libraries. [5] presents an architecture for multi-aspect search based on these ideas.
- In [36] we have presented first steps towards finding views between different theorem prover libraries automatically.

7 Conclusion and Future Work

Summary. In this article, we report on the conclusion of a research objective that seemed quite immediate two decades ago, but was not: the export of a theorem prover library (Isabelle) into a FAIR [55] knowledge exchange format (OMDOC). To make this undertaking feasible at all, both the source and target system had to evolve considerably: Isabelle had to add its Scala and PIDE infrastructure to manage and expose document-oriented information in an instrumentable way, and the OMDOC format had to be re-engineered, extended, and implemented in the MMT system. Of course, the growth of the Isabelle library during this time induced further scalability problems, which we had to solve for our export.

Exports of theorem prover libraries have received substantial attention for the last 10–20 years. Our work is the first comprehensive export for Isabelle: we demonstrate current Isabelle/Scala export technology and explore remaining theoretical and practical challenges.

Even ignoring the potential applications of this particular export, our infrastructure for exporting Isabelle libraries in general will prove beneficial to future improvements to Isabelle itself and to the reuse of Isabelle content in other systems. In fact, the improvements of Isabelle that were needed for our export have already shown benefits for the wider Isabelle community. The *headless PIDE* session and `isabelle dump` tool have become particularly important: we are in personal contact with two different projects to build content-oriented search engines on top of these systems. Another emerging application of this technology is a similar export of Isabelle to Dedukti [7]: this aims at re-checking the Isabelle/AFP and therefore includes proof terms but excludes PIDE document markup.

The current export facility is mostly based on code that is maintained within the Isabelle repository, and thus updated by the core developers. We have already published Isabelle/MMT for Isabelle2019 and Isabelle2020 based on a straight-forward process that users can easily recreate themselves: build MMT within the Isabelle system environment, turn it into an Isabelle component, and use the standard Isabelle release tool to build a stand-alone variant of Isabelle that includes MMT. Users can then rerun our export themselves on the spot (via the `isabelle mmt_import` command). We judge that this makes our Isabelle export the most easily reproducible and maintainable among all existing prover library exports.

Future Work. Besides realizing and scaling up the applications described in Section 6, we want to mention two important avenues for future work:

- The current export does not include proof objects as these would increase its size by an order of magnitude. Instead, we restrict ourselves to the dependency relation induced by the proofs, which already enables many applications, but not, e.g., re-verification of proofs. To obtain scalable proof exports, we must investigate how to shrink the size of the proofs, e.g., by developing a new language for high-level proofs.

- In a similar vein we want to preserve the structure of more high-level declarations – e.g. HOL-type definitions, inductive types. As discussed in Section 4.2, this is supported by MMT and would allow a structurally more similar and thus more understandable export.

References

- 1 Archive of formal proofs (AFP), April 2020. URL: <https://www.isa-afp.org>.
- 2 S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
- 3 Clemens Ballarin. Locales: A module system for mathematical theories. *JAR*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.
- 4 H. Barendregt and H. Geuvers. Proof assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.
- 5 Katja Berčić, Michael Kohlhase, and Florian Rabe. Towards a heterogeneous query language for mathematical knowledge – extended report, 2020. URL: <http://kwarc.info/kohlhase/papers/tetraresearch.pdf>.
- 6 J. Betzendahl and M. Kohlhase. Translating theimps theory library to mmt/omdoc. In F. Rabe, W. Farmer, G. Passmore, and A. Youssef, editors, *Intelligent Computer Mathematics*, volume 11006, pages 7–22. Springer, 2018.
- 7 M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$ -calculus modulo as a universal proof language. In D. Pichardie and T. Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.
- 8 Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2012. doi:10.1007/978-3-642-31374-5_3.
- 9 A. Condoluci, M. Kohlhase, D. Müller, F. Rabe, C. Sacerdoti Coen, and M. Wenzel. Relational Data Across Mathematical Libraries. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 61–76. Springer, 2019.
- 10 T. Gauthier and C. Kaliszyk. Matching concepts across HOL libraries. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 267–281. Springer, 2014.
- 11 M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.
- 12 M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
- 13 F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES conference*, pages 160–174. Springer, 2006.
- 14 Florian Haftmann and Makarius Wenzel. Local theory specifications in Isabelle/Isar. In Stefano Berardi, Ferruccio Damiani, and Ugo de Liguoro, editors, *Types for Proofs and Programs, TYPES 2008*, volume 5497 of *LNCS*. Springer, 2009.
- 15 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- 16 Fulya Horozal, Michael Kohlhase, and Florian Rabe. Extending MKM formats at the statement level. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, number 7362 in *LNAI*, pages 65–80. Springer Verlag, 2012. URL: <http://kwarc.info/kohlhase/papers/mkm12-p2s.pdf>.

- 17 Mihnea Iancu, Michael Kohlhase, Florian Rabe, and Josef Urban. The Mizar Mathematical Library in OMDoc: Translation and applications. *Journal of Automated Reasoning*, 50(2):191–202, 2013. doi:10.1007/s10817-012-9271-4.
- 18 Fabian Immler, Jonas Rädle, and Makarius Wenzel. Virtualization of HOL4 in Isabelle. In John Harrison, John O’Leary, and Andrew Tolmach, editors, *10th International conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *LIPICs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. URL: <http://drops.dagstuhl.de/opus/volltexte/2019/11076/pdf/LIPICs-ITP-2019-21.pdf>.
- 19 Isabelle website, April 2020. URL: <https://isabelle.in.tum.de/website-Isabelle2020>.
- 20 Isabelle/MMT for Isabelle2020, April 2020. URL: https://files.sketis.net/Isabelle_MMT-20200421.
- 21 C. Kaliszzyk, M. Kohlhase, D. Müller, and F. Rabe. A Standard for Aligning Mathematical Concepts. In A. Kohlhase, M. Kohlhase, P. Libbrecht, B. Miller, F. Tompa, A. Naummowicz, W. Neuper, P. Quaresma, and M. Suda, editors, *Work in Progress at CICM 2016*, pages 229–244. CEUR-WS.org, 2016.
- 22 C. Kaliszzyk and J. Urban. HOL(y)hammer: Online ATP service for HOL light. *Mathematics in Computer Science*, 9(1):5–22, 2015.
- 23 Cezary Kaliszzyk and Alexander Krauss. Scalable LCF-style proof translation. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 51–66, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 24 Cezary Kaliszzyk and Florian Rabe. Towards knowledge management for HOL Light. In Stephan Watt, James Davenport, Alan Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics 2014*, number 8543 in *LNCs*, pages 357–372. Springer, 2014. URL: http://kwarc.info/frabe/Research/KR_hollight_14.pdf.
- 25 F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.
- 26 C. Keller and B. Werner. Importing HOL Light into Coq. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, pages 307–322. Springer, 2010.
- 27 M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in *Lecture Notes in Artificial Intelligence*. Springer, 2006.
- 28 M. Kohlhase, D. Müller, S. Owre, and F. Rabe. Making PVS Accessible to Generic Services by Interpretation in a Universal Format. In M. Ayala-Rincon and C. Munoz, editors, *Interactive Theorem Proving*, pages 319–335. Springer, 2017.
- 29 M. Kohlhase and F. Rabe. QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge. *Journal of Formalized Reasoning*, 9(1):201–234, 2016.
- 30 M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.
- 31 Michael Kohlhase. The flexiformalist manifesto. In Andrei Voronkov, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt, and Daniela Zaharie, editors, *14th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012)*, pages 30–36, Timisoara, Romania, 2013. IEEE Press. URL: <http://kwarc.info/kohlhase/papers/synasc13.pdf>.
- 32 Michael Kohlhase and Florian Rabe. Experiences from exporting major proof assistant libraries. submitted, 2020. URL: https://kwarc.info/people/frabe/Research/KR_oafexp_20.pdf.
- 33 Michael Kohlhase and Ioan Şucan. A search engine for mathematical formulae. In Tetsuo Ida, Jacques Calmet, and Dongming Wang, editors, *Proceedings of Artificial Intelligence and Symbolic Computation, AISC’2006*, number 4120 in *LNAI*, pages 241–253. Springer Verlag, 2006. URL: <http://kwarc.info/kohlhase/papers/aisc06.pdf>.

- 34 A. Krauss and A. Schropp. A Mechanized Translation from Higher-Order Logic to Set Theory. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, pages 323–338. Springer, 2010.
- 35 J. Meng and L. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.
- 36 D. Müller, M. Kohlhase, and F. Rabe. Automatically Finding Theory Morphisms for Knowledge Management. In F. Rabe, W. Farmer, G. Passmore, and A. Youssef, editors, *Intelligent Computer Mathematics*, pages 209–224. Springer, 2018.
- 37 D. Müller and F. Rabe. Rapid Prototyping Formal Systems in MMT: Case Studies. In D. Miller and I. Scagnetto, editors, *Logical Frameworks and Meta-languages: Theory and Practice*, pages 40–54, 2019.
- 38 D. Müller, F. Rabe, and C. Sacerdoti Coen. The Coq Library as a Theory Graph. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 171–186. Springer, 2019.
- 39 Dennis Müller, Florian Rabe, Colin Rothgang, and Michael Kohlhase. Representing structural language features in formal meta-languages. submitted, 2020. URL: <http://kwarc.info/kohlhase/submit/cicm20-features.pdf>.
- 40 T. Nipkow and C. Prehofer. Type checking type classes. In *ACM Symp. Principles of Programming Languages*, 1993.
- 41 S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In N. Shankar and U. Furbach, editors, *Automated Reasoning*, volume 4130. Springer, 2006.
- 42 Karl Palmkog, Ahmet Çelik, and Milos Gligoric. piCoq: parallel regression proving for large-scale verification projects. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 344–355. ACM, 2018. doi: 10.1145/3213846.3213877.
- 43 Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In *Logic and Computer Science*, pages 361–386. Academic Press, 1990. URL: <http://www.cl.cam.ac.uk/Research/Reports/TR143-lcp-experience.dvi.gz>.
- 44 Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *Formal Aspects of Computing*, September 2019. Springer, London. doi:10.1007/s00165-019-00492-1.
- 45 A. Pitts. The HOL logic. In M. J. C. Gordon and T. F. Melham, editors, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, pages 191–232. Cambridge University Press, 1993.
- 46 F. Rabe. A Logic-Independent IDE. In C. Benz Müller and B. Woltzenlogel Paleo, editors, *Workshop on User Interfaces for Theorem Provers*, pages 48–60. Elsevier, 2014.
- 47 F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.
- 48 F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.
- 49 Makarius Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP 2014)*, volume 8558 of *LNCS*. Springer, 2014.
- 50 Makarius Wenzel. Interaction with formal mathematical documents in Isabelle/PIDE. In Cezary Kaliszyk, Edwin Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics (CICM 2019)*, volume 11617 of *LNAI*. Springer, 2019. arXiv:1905.01735.
- 51 Makarius Wenzel. *The Isabelle/Isar Implementation*, April 2020. URL: <https://isabelle.in.tum.de/website-Isabelle2020/dist/doc/implementation.pdf>.
- 52 Makarius Wenzel. *Isabelle/jEdit*, April 2020. URL: <https://isabelle.in.tum.de/website-Isabelle2020/dist/doc/jedit.pdf>.

- 53 Makarius Wenzel. *The Isabelle System manual*, April 2020. URL: <https://isabelle.in.tum.de/website-Isabelle2020/dist/doc/system.pdf>.
- 54 Markus Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theys, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*. Springer, 1999.
- 55 Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J. G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A. C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3, 2016. doi:10.1038/sdata.2016.18.

Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules

Jesper Cockx 

Department of Software Technology, TU Delft, The Netherlands

<https://jesper.sikanda.be>

j.g.h.cockx@tudelft.nl

Abstract

Dependently typed languages such as Coq and Agda can statically guarantee the correctness of our proofs and programs. To provide this guarantee, they restrict users to certain schemes – such as strictly positive datatypes, complete case analysis, and well-founded induction – that are known to be safe. However, these restrictions can be too strict, making programs and proofs harder to write than necessary. On a higher level, they also prevent us from imagining the different ways the language could be extended.

In this paper I show how to extend a dependently typed language with user-defined higher-order non-linear rewrite rules. Rewrite rules are a form of equality reflection that is applied automatically by the typechecker. I have implemented rewrite rules as an extension to Agda, and I give six examples how to use them both to make proofs easier and to experiment with extensions of type theory. I also show how to make rewrite rules interact well with other features of Agda such as η -equality, implicit arguments, data and record types, irrelevance, and universe level polymorphism. Thus rewrite rules break the chains on computation and put its power back into the hands of its rightful owner: yours.

2012 ACM Subject Classification Theory of computation \rightarrow Rewrite systems; Theory of computation \rightarrow Equational logic and rewriting; Theory of computation \rightarrow Type theory

Keywords and phrases Dependent types, Proof assistants, Rewrite rules, Higher-order rewriting, Agda

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.2

Supplementary Material The official documentation of rewrite rules in Agda is available in the user manual at <https://agda.readthedocs.io/en/v2.6.1/language/rewriting.html>. The full source code of Agda (including rewrite rules) is available on Github at <https://github.com/agda/agda/>.

1 Introduction

In the tradition of Martin-Löf Type Theory [19], each type former is declared by four sets of rules:

- The **formation rule**, e.g. `Bool : Set`
- The **introduction rules**, e.g. `true : Bool` and `false : Bool`
- The **elimination rules**, e.g. if $P : \text{Bool} \rightarrow \text{Set}$, $b : \text{Bool}$, $pt : P \text{ true}$, and $pf : P \text{ false}$, then `if b then pt else pf : P b`
- The **computation rules**, e.g. `if true then pt else pf = pt` and `if false then pt else pf = pf`

When working in a proof assistant or dependently typed programming language, we usually do not introduce new types directly by giving these rules. That would be very unsafe, as there is no easy way to check that the given rules make sense. Instead, we introduce new rules through schemes that are well-known to be safe, such as strictly positive datatypes, complete case analysis, and well-founded induction.



© Jesper Cockx;

licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 2; pp. 2:1–2:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, users of dependently typed languages or researchers who are experimenting with adding new features to them might find working within these schemes too restrictive. They might be tempted to use `postulate` to simulate the formation, introduction, and elimination rules of new type formers. Yet in intensional type theories there is one thing that cannot be added by using `postulate`: the computation rules.

This paper shows how to extend a dependently typed language with user-defined *rewrite rules*, allowing the user to extend the definitional equality of the language with new computation rules. Concretely, I extend the Agda language [22] with a new option `--rewriting`. When this option is enabled, you can register a proof (or a postulate) $p : \forall x_1 \dots x_n \rightarrow f\ u_1 \dots u_n \equiv v$ (where the \forall quantifies over the free variables $x_1 \dots x_n$ of $u_1 \dots u_n$ and v , and \equiv is Agda's built-in identity type) as a rewrite rule with a pragma `{-# REWRITE p #-}`. From this point on, Agda will automatically reduce instances of the left-hand side $f\ u_1 \dots u_n$ (i.e. for specific values of $x_1 \dots x_n$) to the corresponding instance of v . As a silly example, if $f : A \rightarrow A$ and $p : \forall x \rightarrow f\ x \equiv x$, then the rewrite rule will replace any application $f\ u$ with u , effectively turning f into the identity function $\lambda x \rightarrow x$ (which is the Agda syntax for the lambda term $\lambda x.x$).

Since rewrite rules enable you as the user of Agda to turn propositional (i.e. proven) equalities into definitional (i.e. computational) ones, rewrite rules can be seen as a restricted version of the equality reflection rule from extensional type theory, thus they do not impact logical soundness of Agda directly. However, they can break other important properties of Agda such as confluence of reduction and strong normalization. Checking these properties automatically is outside of the scope of this paper, but some potential approaches are discussed in Sect. 6.

Instead, the main goal of this paper is to specify in detail one possible way to add a general notion of rewrite rules to a real-world dependently typed language. This is meant to serve at the same time as a specification of how rewrite rules are implemented in Agda and also as a guideline how they could be added to other languages.

Contributions

- I define a core type theory based on Martin-Löf's intensional type theory extended with user-defined higher-order non-linear rewrite rules.
- I describe how rewrite rules interact with several common features of dependently typed languages, such as η -equality, data and record types, parametrized modules, proof irrelevance, universe level polymorphism, and constraint solving for metavariables.
- I implement rewrite rules as an extension to Agda and show in six examples how to use them to make writing programs and proofs easier and to experiment with new extensions to Agda.

The official documentation of rewrite rules in Agda is available in the user manual¹. The source code of Agda is available on Github², the code dealing with rewrite rules specifically can be found in the files `Rewriting.hs`³ (418 lines), `NonLinPattern.hs`⁴ (329 lines), `NonLinMatch.hs`⁵ (422 lines), and various other places in the Agda codebase.

¹ <https://agda.readthedocs.io/en/v2.6.1/language/rewriting.html>

² <https://github.com/agda/agda/>

³ <https://github.com/agda/agda/blob/master/src/full/Agda/TypeChecking/Rewriting.hs>

⁴ <https://github.com/agda/agda/blob/master/src/full/Agda/TypeChecking/Rewriting/NonLinPattern.hs>

⁵ <https://github.com/agda/agda/blob/master/src/full/Agda/TypeChecking/Rewriting/NonLinMatch.hs>

Note on the development of rewrite rules in Agda. When the development of rewrite rules in Agda started in 2016, it was expected to be used mainly by type theory researchers to experiment with new computation rules without modifying the implementation of the language itself. For this use case, accepting a large class of rewrite rules is more important than having strong guarantees about (admittedly important) metatheoretical properties such as subject reduction, confluence, or termination, which can be checked by hand if necessary. This is the basis for the rewrite rules as described in the paper.

More recently, Agda users also started using this rewriting facility to enhance Agda's conversion checker with new (proven) equalities, as showcased by the examples in Sect. 2.1 and Sect. 2.2. For this class of users having strong guarantees about subject reduction, confluence and termination is more important. In the future, I would like to extend the support for these users further as outlined in Sect. 6.

Outline of the paper. Sect. 2 consists of examples of how to use rewrite rules to go beyond the usual boundaries set by Agda and define your own computation rules. After these examples, Sect. 3 shows more generally how to add rewrite rules to a dependently typed language, and Sect. 4 shows how rewrite rules interact with other features of Agda. Related work and future work are discussed in Sect. 5 and Sect. 6, and Sect. 7 concludes.

2 Using rewrite rules

With the introduction out of the way, let us start with some examples of things you can do with rewrite rules. I hope at least one example gives you the itch to try rewrite rules for yourself. There are some restrictions on what kind of equality proofs can be turned into rewrite rules, which will be explained later in general. Until then, the examples should give an idea of the kind of things that are possible.

All examples in this section are accepted by Agda 2.6.1 [1]. We start with some basic options and imports. For the purpose of this paper, the two most important ones are the `--rewriting` flag and the import of `Agda.Builtin.Equality.Rewrite`, which are both required to make rewrite rules work. Meanwhile, the `--prop` flag enables Agda's `Prop` universe⁶ [16], which will be used in some of the examples.

```
{-# OPTIONS --rewriting --prop #-}

open import Agda.Primitive
open import Agda.Builtin.Bool
open import Agda.Builtin.Nat
open import Agda.Builtin.List
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
```

The examples in this paper make use of generalizable variables⁷ to avoid writing many quantifiers and make the code more readable.

⁶ <https://agda.readthedocs.io/en/v2.6.1/language/prop.html>

⁷ <https://agda.readthedocs.io/en/v2.6.1/language/generalization-of-declared-variables.html>

2:4 Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules

```
variable
  ℓ ℓ1 ℓ2 ℓ3 ℓ4 : Level
  A B C           : Set ℓ
  P Q             : A → Set ℓ
  x y z           : A
  f g h           : (x : A) → P x
  b b1 b2 b3 : Bool
  k l m n         : Nat
  xs ys zs        : List A
  R               : A → A → Prop
```

We use the following helper function to annotate terms with their types:

```
EI : (A : Set ℓ) → A → A
EI A x = x
```

```
infix 5 EI
syntax EI A x = x ∈ A
```

To avoid reliance on external libraries, we also need two basic properties of equality:

```
cong : (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl

transport : (P : A → Set ℓ) → x ≡ y → P x → P y
transport P refl p = p
```

2.1 Overlapping pattern matching

To start, let us look at a question that is asked by almost every newcomer to Agda: why does `0 + m` compute to `m`, but `m + 0` does not? Similarly, why does `(suc m) + n` compute to `suc (m + n)` but `m + (suc n)` does not? This problem manifests itself for example when trying to prove commutativity of `_+_` (the lack of highlighting is a sign that the code is not accepted by Agda):

```
+comm : m + n ≡ n + m
+comm {m = zero} = refl
+comm {m = suc m} = cong suc (+comm {m = m})
```

Here Agda complains that `n ≠ n + zero`. The problem is usually solved by proving the equations `m + 0 ≡ m` and `m + (suc n) ≡ suc (m + n)` and using an explicit `rewrite`⁸ statement in the proof of `+comm`.

Despite solving the problem, this solution is rather disappointing: if Agda can tell that `0 + m` computes to `m`, why not `m + 0`? During my master thesis, I worked on overlapping computation rules [14] to make this problem go away without adding any explicit `rewrite` statements. By using rewrite rules, we can simulate this solution in Agda. First, we need to prove that the equations we want hold as propositional equalities:

⁸ Agda's `rewrite` keyword should not be confused with rewrite rules, which are added by a `REWRITE` pragma.

```

+zero : m + zero ≡ m
+zero {m = zero} = refl
+zero {m = suc m} = cong suc +zero

+suc : m + (suc n) ≡ suc (m + n)
+suc {m = zero} = refl
+suc {m = suc m} = cong suc +suc

```

Then we mark the equalities as rewrite rules with a **REWRITE** pragma:

```
{-# REWRITE +zero +suc #-}
```

Now the proof of commutativity works exactly as we wrote before:

```

+comm : m + n ≡ n + m
+comm {m = zero} = refl
+comm {m = suc m} = cong suc (+comm {m = m})

```

Without rewrite rules there is **no** way to make this proof go through unchanged: it is essential that `++` computes both on its first and second arguments, but there is no way to define `++` in such a way using Agda's regular pattern matching.

2.2 New equations for neutral terms

Allais, McBride, and Boutillier [2] extend classic functions on lists such as `map`, `++` (concatenation), and `fold` with new equational rules for neutral expressions. In Agda, we can prove these rules and then add them as rewrite rules. For example, here are their rules for `map` and `++`:

```

map : (A → B) → List A → List B
map f [] = []
map f (x :: xs) = (f x) :: (map f xs)

infixr 5 ++
++ : List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

map-id : map (λ x → x) xs ≡ xs
map-id {xs = []} = refl
map-id {xs = x :: xs} = cong (x ::_) map-id

map-fuse : map f (map g xs) ≡ map (λ x → f (g x)) xs
map-fuse {xs = []} = refl
map-fuse {xs = x :: xs} = cong (_ ::_) map-fuse

map-++ : map f (xs ++ ys) ≡ (map f xs) ++ (map f ys)
map-++ {xs = []} = refl
map-++ {xs = x :: xs} = cong (_ ::_) (map-++ {xs = xs})

{-# REWRITE map-id map-fuse map-++ #-}

```

2:6 Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules

These rules look simple, but can be quite powerful. For example, below we show that the expression `map swap (map swap xs ++ map swap ys)` reduces to `xs ++ ys`, without requiring any induction on lists.

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
open _×_

swap : A × B → B × A
swap (x , y) = y , x

test : map swap (map swap xs ++ map swap ys) ≡ xs ++ ys
test = refl
```

To compute the left-hand side of the equation to the right-hand side, Agda makes use of `map-++` ([step₁](#)), `map-fuse` ([step₂](#)), built-in η -equality of `_ × _` ([step₃](#)), the definition of `swap` ([step₄](#)), and finally the `map-id` rewrite rule ([step₅](#)).

```
step1 : map swap (map swap xs ++ map swap ys)
       ≡ map swap (map swap xs) ++ map swap (map swap ys)
step1 = refl

step2 : map swap (map swap xs) ≡ map (λ x → swap (swap x)) xs
step2 = refl

step3 : map (λ x → swap (swap x)) xs ≡ map (λ x → swap (swap (fst x , snd x))) xs
step3 = refl

step4 : map (λ x → swap (swap (fst x , snd x))) xs ≡ map (λ x → (fst x , snd x)) xs
step4 = refl

step5 : map (λ x → (fst x , snd x)) xs ≡ xs
step5 = refl
```

2.3 Higher inductive types

The original motivation for adding rewrite rules to Agda had little to do with adding new computation rules to existing functions as in the previous examples. Instead, its purpose was to experiment with defining higher inductive types [30]. In particular, it was meant as an alternative for people using clever (but horrible) hacks to make higher inductive types compute.⁹

A higher inductive type is similar to a regular inductive type `D` with some additional path constructors, which construct an element of the identity type $a \equiv b$ where $a : D$ and $b : D$. A classic example is the `Circle` type, which has one regular constructor `base` and one path constructor `loop` (note that `Set` in Agda corresponds to `Type` rather than `hSet` from HoTT):

⁹ <https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/>


```

postulate
  Circle : Set
  base   : Circle
  loop   : base ≡ base

postulate
  Circle-elim : (P : Circle → Set ℓ) (base* : P base) (loop* : transport P loop base* ≡ base*)
    → (x : Circle) → P x
  elim-base : ∀ (P : Circle → Set ℓ) base* loop* → Circle-elim P base* loop* base ≡ base*
  {-# REWRITE elim-base #-}

```

To specify the computation rule for `Circle-elim` applied to `loop`, we need the dependent version of `cong`, which is called `apd` in the book [30].

```

apd : (f : (x : A) → P x) (p : x ≡ y) → transport P p (f x) ≡ f y
apd f refl = refl

```

```

postulate
  elim-loop : ∀ (P : Circle → Set ℓ) base* loop* → apd (Circle-elim P base* loop*) loop ≡ loop*
  {-# REWRITE elim-loop #-}

```

Without the rewrite rule `elim-base`, the type of `elim-loop` is not well-formed. So without rewrite rules, it is impossible to even state the computation rule of `Circle-elim` on the path constructor `loop` without adding extra transports that would influence its computational behaviour.

2.4 Quotient types

One of the well-known weak spots of intensional type theory is its poor handling of quotient types. One of the more promising attempts at adding quotients to Agda is by Guillaume Brunerie in the initiality project¹⁰, which uses a combination of rewrite rules and Agda's `Prop` universe. Unlike `Prop` in Coq or `hProp` in HoTT (but like `sProp` in Coq), `Prop` in Agda is a universe of *definitionally* irrelevant propositions, which means any two proofs of a type in `Prop` are definitionally equal.

Before I can show this definition of the quotient type, we first need to define the `Prop`-valued equality type `≐`. We also define its corresponding notion of `transport`, which has to be postulated due to current limitations in the implementation of `Prop`. To make `transportR` compute in the expected way, we add it as a rewrite rule `transportR-refl`.

```

data ≐ {A : Set ℓ} (x : A) : A → Prop ℓ where
  refl : x ≐ x

postulate
  transportR : (P : A → Set ℓ) → x ≐ y → P x → P y
  transportR-refl : transportR {x = x} {y = x} P refl z ≡ z
  {-# REWRITE transportR-refl #-}

```

Note that the rewrite rule `transportR-refl` is non-linear in its two implicit arguments x and y .

¹⁰<https://github.com/guillaumebrunerie/initiality/blob/reflection/quotients.agda>

2:8 Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules

Now we are ready to define the quotient type `__//__`. Given a type A and a `Prop`-valued relation $R : A \rightarrow A \rightarrow \text{Prop}$, the type $A // R$ consists of elements `proj x` where $x : A$, and `proj x` is equal to `proj y` if and only if $R x y$ holds.

```
postulate
  __//__ : (A : Set ℓ) (R : A → A → Prop) → Set ℓ
  proj   : A → A // R
  quot   : R x y → proj {R = R} x ≐ proj {R = R} y
```

The elimination principle `//-elim` allows us to define functions that extract an element of A from a given element of $A // R$, provided a proof `quot*` that the function respects the equality on $A // R$. The computation rule `//-beta` allows `//-elim` to compute when it is applied to a `proj x`.

```
//-elim : (P : A // R → Set ℓ) (proj* : (x : A) → P (proj x))
  → (quot* : {x y : A} (r : R x y) → transportR P (quot r) (proj* x) ≐ proj* y)
  → (x : A // R) → P x
//-beta : {R : A → A → Prop} (P : A // R → Set ℓ) (proj* : (x : A) → P (proj x))
  → (quot* : {x y : A} (r : R x y) → transportR P (quot r) (proj* x) ≐ proj* y)
  → {u : A} → //-elim P proj* quot* (proj u) ≡ proj* u
{-# REWRITE //-beta #-}
```

Compared to the more standard way of defining the quotient type as a higher inductive type, this definition behaves better with respect to definitional equality: the argument `quot*` to the eliminator is definitionally irrelevant, so it does not matter what equality proof we give. Consequently, there is no need to add an additional constructor to truncate the quotient type.

2.5 Exceptional type theory

First-class exceptions are a common feature of object-oriented programming languages such as Java, but in the world of pure functional languages they are usually frowned upon. However, recently Pédrot and Tabareau have proposed an extension of Coq with first-class exceptions [23]. With the exceptional power of rewrite rules, we can also encode (part of) their system in Agda.

First, we postulate a type `Exc` with any kinds of exceptions we might want to use (here we just have a single `runtimeException` for simplicity). We then add the possibility to `raise` an exception, producing an element of an arbitrary type A .

```
postulate
  Exc : Set
  runtimeException : Exc
  raise : Exc → A
```

Note that `raise` makes the type theory inconsistent. In their paper, Pédrot and Tabareau show how to build a safe version of exceptions on top of this system, using parametricity to enforce that all exceptions are caught locally. Here that part is omitted for brevity.

By adding the appropriate rewrite rules for each type former, we can ensure that exceptions are propagated appropriately. For positive types such as `Nat`, exceptions are propagated outwards, while for negative types such as function types, exceptions are propagated inwards.

postulate

```
raise-suc : {e : Exc} → suc (raise e) ≡ raise e
raise-fun : {e : Exc} → raise {A = (x : A) → P x} e ≡ λ x → raise {A = P x} e
{-# REWRITE raise-suc raise-fun #-}
```

To complete the system, we add the ability to `catch` exceptions at specific types. This takes the shape of an eliminator with one additional method for handling the case where the element under scrutiny is of the form `raise e`.

postulate

```
catch-Bool : (P : Bool → Set ℓ) (pt : P true) (pf : P false)
  → (h : ∀ e → P (raise e)) → (b : Bool) → P b

catch-true : ∀ (P : Bool → Set ℓ) pt pf h → catch-Bool P pt pf h true ≡ pt
catch-false : ∀ (P : Bool → Set ℓ) pt pf h → catch-Bool P pt pf h false ≡ pf
catch-exc : ∀ (P : Bool → Set ℓ) pt pf h e → catch-Bool P pt pf h (raise e) ≡ h e
{-# REWRITE catch-true catch-false catch-exc #-}
```

As shown by this example, rewrite rules can be used to extend Agda with new primitive operations, including ones that compute according to the type of their arguments. Currently the user has to add new rewrite rules manually for each datatype and function symbol, so using this in practice is quite tedious. In the future, it might be possible to leverage Agda's reflection framework to generate these rewrite rules automatically.

2.6 Observational equality

Rewrite rules also allow us to define type constructors that compute according to the type they are applied to. This is a core part of observational type theory (OTT) [3]. OTT replaces the usual identity type with an observational equality type (here called `≅`) that computes according to the type of the elements being compared. For example, an equality proof between pairs of type $(a, b) \cong (c, d)$ is a pair of proofs, one of type $a \cong c$ and one of type $b \cong d$.

Below, I show how to extend Agda with a fragment of OTT. Since OTT has a proof-irrelevant equality type, I use Agda's `Prop` to get the same effect. First, we need some basic types in `Prop`:

```
record ⊤ {ℓ} : Prop ℓ where constructor tt

data ⊥ {ℓ} : Prop ℓ where

record _∧_ (X : Prop ℓ1) (Y : Prop ℓ2) : Prop (ℓ1 ⊔ ℓ2) where
  constructor _,_
  field
    fst : X
    snd : Y
  open _∧_
```

The `open` statement makes the constructor and the fields of the records available in the remainder of the module.

The central type of OTT is observational equality `≅`, which should compute according to the types of the elements being compared. Here I give the computation rules for `Bool` and for function types:

2:10 Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules

```

infix 6 _≅_
postulate
  _≅_ : {A : Set ℓ1} {B : Set ℓ2} → A → B → Prop (ℓ1 ∪ ℓ2)

postulate
  refl-Bool   : (Bool ≅ Bool) ≡ ⊤
  refl-true   : (true  ≅ true) ≡ ⊤
  refl-false  : (false ≅ false) ≡ ⊤
  conflict-tf : (true  ≅ false) ≡ ⊥
  conflict-ft : (false ≅ true)  ≡ ⊥
{-# REWRITE refl-Bool refl-true refl-false conflict-tf conflict-ft #-}

postulate
  cong-Π : ((x : A) → P x) ≅ ((y : B) → Q y)
          ≡ (B ≅ A) ∧ ((x : A)(y : B) → y ≅ x → P x ≅ Q y)
  cong-λ : {A : Set ℓ1} {B : Set ℓ2} {P : A → Set ℓ3} {Q : B → Set ℓ4}
          → (f : (x : A) → P x) (g : (y : B) → Q y)
          → ((λ x → f x) ≅ (λ y → g y)) ≡ ((x : A) (y : B) (x ≅ y : x ≅ y) → f x ≅ g y)
{-# REWRITE cong-Π cong-λ #-}

```

According to `cong-Π`, an equality proof between function types computes to a pair of equality proofs between the domains and the codomains respectively. Though not necessary, it is convenient to swap the sides of the equality proofs in contravariant positions ($B \equiv A$ and $y \equiv x$). Meanwhile, an equality proof between two functions computes to an equality proof between the functions applied to heterogeneously equal variables $x : A$ and $y : B$.

To reason about equality proofs, OTT adds two more notions: **coercion** and **cohesion**. Coercion `_[_]` transforms an element from one type to the other when both types are observationally equal, and cohesion `_||_` states that coercion is computationally the identity.

```

infix 10 _[_] _||_

```

```

postulate
  _[_] : A → (A ≅ B) → B
  _||_ : (x : A) (Q : A ≅ B) → (x ∈ A) ≅ (x [ Q ] ∈ B)

```

Here the \in annotations are just there to help Agda's type inference algorithm.

Again, we need rewrite rules to make sure coercion computes in the right way when applied to specific type constructors. On the other hand, We do not need rewrite rules for coherence since the result is of type `_ ≅ _` which is a `Prop`, so the proof is anyway irrelevant.

Coercing an element from `Bool` to `Bool` is easy.

```

postulate
  coerce-Bool : (Bool ≅ Bool : Bool ≅ Bool) → b [ Bool ≅ Bool ] ≡ b
{-# REWRITE coerce-Bool #-}

```

To coerce a function from $(x : A) \rightarrow P x$ to $(y : B) \rightarrow Q y$ we need to:

1. Coerce the input from $y : B$ to $x : A$
2. Apply the function to get an element of type $P x$
3. Coerce the output back to an element of $Q y$

In the last step, we need to use coherence to show that x and y are (heterogeneously) equal.

postulate

```

coerce-Π : {A : Set ℓ1} {B : Set ℓ2} {P : A → Set ℓ3} {Q : B → Set ℓ4} {f : (x : A) → P x}
  → (ΠAP≅ΠBQ : ((x : A) → P x) ≅ ((y : B) → Q y))
  → f [ ΠAP≅ΠBQ ]
  ≡ (λ (y : B) →
    let B≅A = fst ΠAP≅ΠBQ
        x    = y [ B≅A ]
        Px≅Qy = snd ΠAP≅ΠBQ x y (||_ {B = A} y B≅A)
    in f x [ Px≅Qy ] ∈ Q y)
{-# REWRITE coerce-Π #-}

```

Here the syntax $\{B = A\}$ instantiates the implicit argument B of $||_$ to the value A .

Of course this is just a fragment of the whole system, but implementing all of OTT would go beyond the scope of this paper. In principle, observational equality can be used as a full replacement for Agda's built-in equality type. So rewrite rules are even powerful enough to experiment with replacements for core parts of Agda.

3 Type theory with user-defined rewrite rules

In the previous section, I gave several examples of how to use rewrite rules in Agda to make programming and proving easier and to experiment with new extensions to type theory. The next two sections go into the details of how rewrite rules work in general.

Instead of starting with a complex language like Agda, I start with a small core language and gradually extend it by adding more features to the rewriting machinery step by step. In the next section, I will extend this language with other features that you are used to from Agda. The full rules of the language can be found in Appendix A.

3.1 Syntax

We use a simplified version of the internal syntax used by Agda [22]. The syntax has five constructors: variables, function symbols, lambdas, pi types, and universes.

$$\boxed{u, v, A, B} ::= \begin{array}{l} x \bar{u} \quad (\text{variable applied to zero or more arguments}) \\ f \bar{u} \quad (\text{function symbol applied to zero or more arguments}) \\ \lambda x. u \quad (\text{lambda abstraction}) \\ (x : A) \rightarrow B \quad (\text{dependent function type}) \\ \mathbf{Set}_i \quad (i\text{th universe}) \end{array} \tag{1}$$

As in the internal syntax of Agda, there is no way to represent a β -redex in this syntax. Instead, substitution $\boxed{u\sigma}$ is defined to eagerly reduce β -redexes on the fly. Since terms are always in β -normal form, our rewrite system is a HRS (Higher-Order Rewrite system) in the spirit of Mayr and Nipkow [20].

Contexts are right-growing lists of variables annotated with their types.

$$\Gamma, \Delta ::= \cdot \quad (\text{empty context}) \\
\quad | \Gamma(x : A) \quad (\text{context extension}) \tag{2}$$

Patterns $\boxed{p, q}$ share their syntax with regular terms, but must satisfy some additional restrictions. To start with, the only allowed patterns are unapplied variables x and applications of function symbols to other patterns $\mathbf{f} \bar{p}$. This allows us for example to declare rewrite rules like `plus x zero \rightarrow x` and `plus x (suc y) \rightarrow suc (x + y)`.

3.2 Declarations

There are two kinds of declarations: function symbols (corresponding to a `postulate` in Agda) and rewrite rules (corresponding to a `postulate` together with a `{-# REWRITE #-}` pragma).

$$\boxed{\mathbf{d}} ::= \begin{array}{l} \mathbf{f} : A \quad \text{(function symbol)} \\ | \quad \forall \Delta. \mathbf{f} \bar{p} : A \rightarrow v \quad \text{(rewrite rule)} \end{array} \quad (3)$$

When the user declares a new rewrite rule, the following properties are checked:

Linearity. Each variable in Δ must occur exactly once in the pattern \bar{p} (this will later be relaxed to “at least once”).

Well-typedness. The left- and right-hand side of the rewrite rule must be well-typed and have the same type, i.e. $\Delta \vdash \mathbf{f} \bar{p} : A$ and $\Delta \vdash v : A$.

Neutrality. The left-hand side of the rewrite rule should be neutral, i.e. it should not reduce.

The first restriction ensures that all variables of a rewrite rule are bound by the left-hand side. This ensures that reduction can never introduce variables that are not in scope, which would break well-scopedness of expressions. The second restriction ensures that applying a rewrite rule does not change the type of a well-typed expression.¹¹ It is possible to go without the third restriction, but in practice this would mean that the rewrite rule would never be applied.¹²

Requiring rewrite rules to be well-typed has in some cases the unfortunate side-effect of introducing non-linearity where it is not really necessary, for example when defining the computation rule of the `J` eliminator as a rewrite rule. This non-linearity slows down the reduction unnecessarily and greatly complicates confluence checking. It would be interesting to investigate how to remove this unnecessary non-linearity, e.g. as proposed by Blanqui [8].

3.3 Reduction and matching

To reduce a term $\mathbf{f} \bar{u}$, we look at the rewrite rules with head symbol \mathbf{f} to see if any of them apply. In the rule below and all rules in the future, we assume a fixed global signature Σ containing all (preceding) declarations.

$$\frac{(\forall \Delta. \mathbf{f} \bar{p} : A \rightarrow v) \in \Sigma \quad [\bar{u} // \bar{p}] \Rightarrow \sigma}{\mathbf{f} \bar{u} \rightarrow v \sigma} \quad (4)$$

Matching a term u against a pattern p $\boxed{[u // p] \Rightarrow \sigma}$ (or $\boxed{[\bar{u} // \bar{p}] \Rightarrow \sigma}$ for matching a list of terms against a list of patterns) produces – if it succeeds – a substitution σ . In contrast to the first-match semantics of clauses of a regular definition by pattern matching, all rewrite rules are considered in parallel, so there is no need for separate notion of a failing match.

¹¹To prove type preservation we also need confluence of reduction, see the future work section for more details.

¹²If the rewrite system is globally confluent and strongly normalizing, it does not matter that we never apply a certain rewrite rule. Global confluence ensures that even if we apply a different rewrite rule, the result will still be the same, and strong normalization ensures that termination does not depend on the choice of rewrite rule either. Hence as a user one does not have to worry about the precise rewrite strategy implemented by Agda, but only about confluence and termination of the rewrite system.

$$\begin{array}{c}
\frac{}{[u // x] \Rightarrow [u / x]} \qquad \frac{u \longrightarrow^* \mathbf{f} \bar{v} \quad [\bar{v} // \bar{p}] \Rightarrow \sigma}{[u // \mathbf{f} \bar{p}] \Rightarrow \sigma} \\
\frac{}{[\cdot // \cdot] \Rightarrow []} \qquad \frac{[u // p] \Rightarrow \sigma_1 \quad [\bar{u} // \bar{p}] \Rightarrow \sigma_2}{[u; \bar{u} // p; \bar{p}] \Rightarrow \sigma_1 \uplus \sigma_2}
\end{array}$$

■ **Figure 1** Basic rules for the matching algorithm used for rewriting.

The basic matching algorithm is defined by the rules in Fig. 1. Matching a term against a pattern variable produces a substitution that assigns the given value to the variable. Matching an expression against a pattern $\mathbf{f} \bar{p}$ evaluates the expression until it becomes of the form $\mathbf{f} \bar{v}$ (here \longrightarrow^* is the reflexive and transitive closure of \longrightarrow). It then recursively matches the arguments \bar{v} against the patterns \bar{p} , combining the results of each match by taking the disjoint union $\sigma_1 \uplus \sigma_2$. Since matching can reduce the term being matched, matching and reduction are mutually recursive.

3.4 Higher-order matching

With the basic set of rewrite rules introduced in the previous section, we can already declare a surprisingly large number of rewrite rules for first-order algebraic structures. From the examples in Sect. 2, it handles all of Sect. 2.1, rules `map-fuse` and `map-++` from Sect. 2.2, all of Sect. 2.3, rule `//-beta` from Sect. 2.4, rules `catch-true`, `catch-false`, and `catch-exc` from Sect. 2.5, and the rules dealing with `Bool` in Sect. 2.6.

Most of the examples that are not yet handled use λ and/or function types in the pattern of a rewrite rule. This brings us to the issue of *higher-order matching*.¹³ To support higher-order matching, we extend the pattern syntax with the following patterns:

- A lambda pattern $\boxed{\lambda x. p}$
- A function type pattern $\boxed{(x : p) \rightarrow q}$
- A bound variable pattern $\boxed{y \bar{p}}$, where y is a variable bound locally in the pattern by a lambda or function type
- A pattern variable $\boxed{x \bar{y}}$ applied to locally bound variables

During matching we must keep the (rigid) bound variables separate from the (flexible) pattern variables. For this purpose, the algorithm keeps a list Φ of all rigid variables. This list is not touched by any of the rules of Fig. 1, but any variables bound by a λ or a function type are added to it.

The extended matching rules for higher-order patterns are given in Fig. 2. Note the strong similarity between the third rule and the rule for matching a function symbol \mathbf{f} . This is not a coincidence: both function symbols and bound variables act as rigid symbols that can be matched against. The first three rules in Fig. 2 extend the pattern syntax to allow for bound variables in patterns, and allow for rules such as `map-id`: `map` $(\lambda x \rightarrow x) xs \equiv xs$. However, alone they do not yet constitute true higher-order matching (such as used in rules `raise-fun`, `cong- Π` , and `cong- λ`). For this we also consider *pattern variables* applied to zero or more

¹³ See also <https://github.com/agda/agda/issues/1563> for more examples where higher-order matching is needed.

$$\begin{array}{c}
\frac{u \longrightarrow^* \lambda x. v \quad \Phi, x \vdash [v // p] \Rightarrow \sigma}{\Phi \vdash [u // \lambda x. p] \Rightarrow \sigma} \\
\\
\frac{A \longrightarrow^* (x : B) \rightarrow C \quad \Phi \vdash [B // p] \Rightarrow \sigma_1 \quad \Phi, x \vdash [C // q] \Rightarrow \sigma_2}{\Phi \vdash [A // (x : p) \rightarrow q] \Rightarrow \sigma_1 \uplus \sigma_2} \\
\\
\frac{u \longrightarrow^* x \bar{v} \quad x \in \Phi \quad \Phi \vdash [\bar{v} // \bar{p}] \Rightarrow \sigma}{\Phi \vdash [u // x \bar{p}] \Rightarrow \sigma} \qquad \frac{x \notin \Phi \quad FV(v) \cap \Phi \subseteq \bar{y}}{\Phi \vdash [v // x \bar{y}] \Rightarrow [(\lambda \bar{y}. v) / x]}
\end{array}$$

■ **Figure 2** Rules for higher-order pattern matching.

arguments. Allowing arbitrary patterns as arguments to pattern variables is well known to make matching undecidable, so we restrict patterns to Miller’s pattern fragment [21] by requiring pattern variables to be applied to distinct bound variables. Matching against a pattern variable in the Miller fragment is implemented by the fourth rule in Fig. 2. Since all the arguments of x are variables, we can construct the lambda term $\lambda \bar{y}. v$. To avoid having out-of-scope variables in the resulting substitution, the free variables in v are checked to be included in \bar{y} , otherwise matching fails.

3.5 Eta equality

The attentive reader may have noticed a flaw in the matching for λ -patterns: it does not respect η -equality. With η -equality for functions, any term $u : (x : A) \rightarrow B$ can always be expanded to $\lambda x. u x$, so it should also match a pattern $\lambda x. p$. A naive attempt to add η -equality would be to η -expand on the fly whenever we match something against a λ -pattern:

$$\frac{\Phi, x \vdash [u x // p] \Rightarrow \sigma}{\Phi \vdash [u // \lambda x. p] \Rightarrow \sigma} \tag{5}$$

This is however not enough to deal with η -equality in general. It is possible that the pattern itself is underapplied as well, e.g. when we match a term of type $(x : A) \rightarrow B$ against a pattern $\mathbf{f} \bar{p}$ or $x \bar{p}$. For example, when we have symbols $\mathbf{f} : (\mathbf{Nat} \rightarrow \mathbf{Nat}) \rightarrow \mathbf{Bool}$ and $\mathbf{g} : \mathbf{Nat} \rightarrow \mathbf{Nat}$ with rewrite rules $\mathbf{f} \mathbf{g} \longrightarrow \mathbf{true}$ and $\forall (x : \mathbf{Nat}). \mathbf{g} x \longrightarrow x$, then we want $\mathbf{f} (\lambda x. x)$ to reduce to \mathbf{true} , but with the above rule matching is stuck on the problem $[\lambda x. x // \mathbf{g}]$.

To respect eta equality for functions and record types, we need to make matching *type-directed*. We also need contexts with the types of the free and bound variables. Thus we extend the matching judgement to $\Gamma; \Phi \vdash [u : A // p] \Rightarrow \sigma$ where A is the type of u (note: not necessarily the same as the type of p) and Γ and Φ are now contexts of pattern variables and bound variables respectively.

The type information is used by the matching algorithm to do on-the-fly η -expansion of functions whenever the type is (or computes to) a function type:

$$\frac{A \longrightarrow^* (x : B) \rightarrow C \quad \Gamma; \Phi(x : B) \vdash [u x : C // p x] \Rightarrow \sigma}{\Gamma; \Phi \vdash [u : A // p] \Rightarrow \sigma} \tag{6}$$

Here $p x$ is only defined if the result is actually a pattern, otherwise the rule cannot be applied.

Having access to the type of the expression being matched is not only useful for η -equality of functions, but also for non-linear patterns (Sect. 3.6), η -equality for records (Sect. 4.1), and irrelevance (Sect. 4.4). While type-directed matching might slow down the reduction in some cases, I believe in many cases the benefits outweigh this disadvantage. Moreover, using irrelevance to avoid unnecessary conversion checks might even make up for the lost performance.

3.6 Non-linearity and conditional rewriting

Sometimes it is desirable to declare rewrite rules with *non-linear* patterns, i.e. where a pattern variable occurs more than once. As an example, this allows us to postulate an equality proof `trustMe` : $(x\ y : A) \rightarrow x \equiv y$ with a rewrite rule `trustMe` $x\ x \equiv \text{refl}$. This can be used in a similar way to Agda’s built-in `primTrustMe`¹⁴. Another example where non-linearity is used is the rule `transportR-refl` from the example in Sect. 2.4, which is non-linear in its two implicit arguments x and y .¹⁵

Non-linear matching is a specific instance of *conditional rewriting*. For example, the non-linear rule `trustMe` $x\ x \equiv \text{refl}$ can be seen equivalently as the linear rule `trustMe` $x\ y \equiv \text{refl}$ with an extra condition $x = y : A$.

Using conditional rewriting, we can not only allow non-linear patterns but also patterns that contain arbitrary terms that do not fall in the pattern fragment. Like for non-linear rules, these “non-pattern” parts of the pattern are replaced by a fresh variable and a constraint that enforces this variable to be definitionally equal to the actual term. The only restriction is that all variables must be bound at least once in a pattern position.

This use of conditional rewriting is similar to inaccessible patterns (also known as dot patterns in Agda) used in dependent pattern matching, with the important difference that inaccessible patterns are guaranteed to match by the type system, while the constraints for conditional rewriting have to be *checked*.

To check the equality constraints of conditional rewrite rules, the matching algorithm needs to decide whether two given terms are definitionally equal. This means reduction and matching are now mutually recursive with conversion checking.¹⁶ We make use of a type-directed conversion judgement $\Gamma \vdash u = v : A$ (see the appendix for the full conversion rules). The new judgement form of matching is now $\Gamma; \Phi \vdash [v : A // p] \Rightarrow \sigma; \Psi$, where Ψ is a set of constraints of the form $\Phi \vdash u \stackrel{?}{=} v$. We extend the matching algorithm with the ability to generate new constraints:

$$\frac{}{\Gamma; \Phi \vdash [v : A // p] \Rightarrow []; \{\Phi \vdash v \stackrel{?}{=} p : A\}} \quad (7)$$

¹⁴<https://agda.readthedocs.io/en/v2.6.1/language/built-ins.html#primtrustme>

¹⁵It also needs irrelevance for `Prop`, see Sect. 4.4 for more details.

¹⁶To actually change the implementation of Agda to make the matching depend on conversion checking took quite some effort (see <https://github.com/agda/agda/pull/3589>). The reason for this difficulty was that reduction and matching are running in one monad `ReduceM`, while conversion was running in another monad `TCM` (short for “type-checking monad”). The new version of the conversion checker is polymorphic in the monad it runs in. This means the same piece of code implements at the same time a pure, declarative conversion checker and a stateful constraint solver.

All other rules just gather the set of constraints, taking the union whenever matching produces multiple sub-problems. When matching concludes, the constraints are checked before the rewrite rule is applied:

$$\frac{\begin{array}{l} f : \Gamma \rightarrow A \in \Sigma \quad (\forall \Delta. f \bar{p} : B \rightarrow v) \in \Sigma \\ [\bar{u} : \Gamma[\bar{u}] // \bar{p}] \Rightarrow \sigma; \Psi \quad \forall (\Phi \vdash v \stackrel{?}{=} p : A) \in \Psi. \Phi \vdash v = p\sigma : A \end{array}}{f \bar{u} \rightarrow v\sigma} \quad (8)$$

When checking a constraint we apply the final substitution σ to the pattern p but not to the term v or the type A . This makes sense because the term being matched does not contain any pattern variables in the first place (and neither does its type).

4 Interaction with other features

Adding rewrite rules to an existing language such as Agda is quite an undertaking. Rewrite rules often interact with other features in a non-trivial matter, and it takes work to resolve these interactions in a satisfactory way. In this section, I describe the interaction of rewrite rules with several other features of Agda: record types with eta equality, datatypes, parametrized modules, definitional irrelevance, universe polymorphism, and constraint solving.

4.1 Eta equality for records

Agda has η -equality not just for function types, but also for record types. For example, any term $u : A \times B$ is definitionally equal to $(\mathbf{fst} \ u, \mathbf{snd} \ u)$. Since η -equality of records is a core part of Agda, we extend the matching algorithm to deal with it.¹⁷ As for η -equality of functions, we make use of the type of the expression to η -expand terms and patterns during matching.

Let $R : \mathbf{Set}_i$ be a record type with fields $\pi_1 : A_1, \dots, \pi_n : A_n$. We have the following matching rule:

$$\frac{\Gamma; \Phi \vdash [\pi_i \ u : A_i[\overline{\pi_j \ u / \pi_j}^{j < i}] // \pi_i \ p] \Rightarrow \sigma \quad (i = 1 \dots n)}{\Gamma; \Phi \vdash [u : R // p] \Rightarrow \sigma} \quad (9)$$

Since records can be dependent, each type A_i may depend on the previous fields π_1, \dots, π_{i-1} , so we need to substitute the concrete values $\pi_j \ u$ for π_j in A_i for each $j < i$.

In the case where $n = 0$, this rule says that a term of the unit record type \top (with no fields) matches any pattern. So the matching algorithm even handles the notorious η -unit types.

4.2 Datatypes and constructors

An important question is how rewrite rules interact with datatypes such as `Nat`, `List`, and `__≡__`. Can we simply add rewrite rules to (type and/or term) constructors? The answer is actually a bit more complicated.

¹⁷See <https://github.com/agda/agda/issues/2979> and <https://github.com/agda/agda/issues/3335>.

If we allow rewriting of datatype constructors, we could (for example) postulate an equality proof of type `Nat ≡ Bool` and register it as a rewrite rule. However, this would mean `zero : Bool`, violating an important internal invariant of Agda that any time we have `c \bar{u} : D` for a constructor `c` and a datatype `D`, `c` is actually a constructor of `D`.¹⁸ For this reason, it is not allowed to have rewrite rules on datatypes or record types.

For constructors of datatypes there is no a priori reason why they cannot have rewrite rules attached to them. This would actually be useful to define a “definitional quotient type” where some of the constructors may compute. Unfortunately, there is another problem: internally, Agda does not store the constructor arguments corresponding to the parameters of the datatype. For example, the constructors `[]` and `_::_` of the `List A` type do not store the type `A` as an argument. This is important for efficient representation of parametrized datatypes. However, this means that rewrite rules that match on constructors cannot match against arguments in those positions, or bind pattern variables in them.

When a rewrite rule is added with a constructor as the head symbol, we have to take care that the rewrite rule is not applied too generally. For example, a rewrite rule for `[] : List Nat` should not be applied to `[] : List A` where `A ≠ Nat`.¹⁹ To avoid unwanted reductions like these, it is only allowed to add a rewrite rule to a constructor if the parameters are *fully general*, i.e. they must be distinct variables. This ensures that rewrite rules are only applied to terms whose type matches the type of the rewrite rule.

4.3 Parametrized modules and “where” blocks

A parametrized module is a collection of declarations parametrized over a common telescope Γ . In one sense, parametrized modules can be thought of as λ -lifting all the definitions inside the module: if a module with parameters Γ contains a definition of `f : A`, then the real type of `f` is $\Gamma \rightarrow A$. But this does not quite capture the intuition that definitions inside a parametrized module should be *parametric* in the parameters. So module parameters should be treated as rigid symbols like postulates rather than as flexible variables.

For this reason, module parameters play a double role on the left-hand side of a rewrite rule:

- As long as the parameter is in scope (i.e. inside the module), it has to match “on the nose” (i.e. it cannot be instantiated by matching).
- Once the parameter goes out of scope (i.e. outside of the module), it is treated as a regular pattern variable that can be instantiated by matching.

For example, inside a module parametrized over `n : Nat`, a rewrite rule `f n → zero` only applies to terms definitionally equal to `f n`. On the other hand, outside of the module the rewrite rule applies to any expression of the form `f u`.

This intuition of module parameters as rigid symbols also applies to Agda’s treatment of `where` blocks, which are nothing more than modules parametrized over the pattern variables of the clause (you can even give a name to the `where` module using the `module M where` syntax²⁰). Here a rewrite rule declared in a `where` block should only apply for the specific arguments to the function that are used in the clause, not those of a recursive call²¹.

¹⁸ See <https://github.com/agda/agda/issues/3846>.

¹⁹ See <https://github.com/agda/agda/issues/3211>.

²⁰ <https://agda.readthedocs.io/en/v2.6.1/language/let-and-where.html#where-blocks>

²¹ <https://github.com/agda/agda/issues/1652>

4.4 Irrelevance and Prop

Another feature of Agda is *definitional irrelevance*, which comes in the two flavours of irrelevant function types $.A \rightarrow B^{22}$ and the universe **Prop** of definitionally proof-irrelevant propositions²³. For rewrite rules with irrelevant parts in their patterns matching should never fail because this would mean a supposedly irrelevant term is not actually irrelevant. However, it should still be allowed to bind a variable in an irrelevant position, since we might want to use that variable in (irrelevant positions of) the right-hand side.²⁴ This means in irrelevant positions we allow:

1. pattern variables $x \bar{y}$ where \bar{y} are all the bound variables in scope, and
2. arbitrary terms u that do not bind any variables.

Both of these will always match any given term: the former because \bar{y} is required to consist of all bound variables, and the latter because two irrelevant terms are always considered equal by the conversion checker. However, only the former can bind a variable.

Together with the ability to have non-linear patterns, this allows us to have rewrite rules such as `transportR – refl` : `transportR P refl x ≡ x` where `transportR` : $(P : A \rightarrow \text{Set}_\ell) \rightarrow x \doteq y \rightarrow P x \rightarrow P y$ and $x \doteq y$ is the equality type in **Prop**. The constructor `refl` here is irrelevant, so this rule does not actually match against the constructor `refl`. Instead, Agda checks that the two arguments x and y are definitionally equal, and applies the rewrite rule if this is the case.

4.5 Universe level polymorphism

Universe level polymorphism allows Agda programmers to write definitions that are polymorphic in the universe level of a type parameter. Since the type **Level** of universe levels is a first-class type in Agda, it interacts natively with rewrite rules: patterns can bind variables of type **Level** just as any other type. This allows us for example to define rewrite rules such as `map – id` that work on level-polymorphic lists.

The type **Level** supports two operations `lsuc` : **Level** \rightarrow **Level** and `_□_` : **Level** \rightarrow **Level** \rightarrow **Level**. These operations have a complex equational structure: `_□_` is associative, commutative, and idempotent, and `lsuc` distributes over `_□_`, just to name a few of the laws. This causes trouble when a rewrite rule matches against one of these symbols: how should it determine whether a given level matches $a \square b$ when `_□_` is commutative?²⁵ For this reason it is not allowed to have rewrite rules that match against `lsuc` or `_□_`.

This restriction on patterns of type **Level** seems reasonable enough, but it is often not satisfied by rewrite rules that match on function types – like the `cong–Π` rule we used in the encoding of observational type theory (Sect. 2.6). The problem is that if $A : \text{Set}_{\ell_1}$ and $B : \text{Set}_{\ell_2}$, then the function type $(x : A) \rightarrow B$ has type $\text{Set}_{\ell_1 \sqcup \ell_2}$, so there is no sensible position to bind the variables ℓ_1 and ℓ_2 .

To allow rewrite rules such as `cong–Π`, we need to find a different position where these variables of type **Level** can be bound. In the internal syntax of Agda, function types $(x : A) \rightarrow B$ are annotated with the sorts of A and B . So the “real” function type of Agda

²² <https://agda.readthedocs.io/en/v2.6.1/language/irrelevance.html>

²³ <https://agda.readthedocs.io/en/v2.6.1/language/prop.html>

²⁴ See <https://github.com/agda/agda/issues/2300>.

²⁵ Issue #2090 (<https://github.com/agda/agda/issues/2090>) and issue #2299 (<https://github.com/agda/agda/issues/2299>) show some of the things that would go wrong.

is of the form $(x : A : \text{Set}_{\ell_1}) \rightarrow (B : \text{Set}_{\ell_2})$. This means that if we allow rewrite rules to bind pattern variables in these hidden annotations, we are saved.²⁶ The matching rule for function types now becomes:

$$\frac{\Gamma; \Phi \vdash [A : \text{Set } \ell_1 // p] \Rightarrow \sigma_1; \Psi_1 \quad \Gamma; \Phi \vdash [\ell_1 : \text{Level } // q] \Rightarrow \sigma_2; \Psi_2 \quad \Gamma; \Phi(x : A) \vdash [B : \text{Set } \ell_2 // r] \Rightarrow \sigma_3; \Psi_3 \quad \Gamma; \Phi \vdash [\ell_2 : \text{Level } // s] \Rightarrow \sigma_4; \Psi_4}{\Gamma; \Phi \vdash [(x : A : \text{Set } \ell_1) \rightarrow (B : \text{Set } \ell_2) // (x : p : \text{Set}_q) \rightarrow (r : \text{Set}_s)] \Rightarrow (\sigma_1 \uplus \sigma_2 \uplus \sigma_3 \uplus \sigma_4); (\Psi_1 \cup \Psi_2 \cup \Psi_3 \cup \Psi_4)} \quad (10)$$

Thanks to this rule, also the universe-polymorphic version of the rewrite rules in Sect. 2.6 are accepted by Agda.

4.6 Metavariables and constraint solving

To automatically fill in the values of implicit arguments, Agda inserts *metavariables* as their placeholders. These metavariables are then solved during typechecking by the constraint solver. A full description of Agda’s constraint solver is out of the scope of this paper, but let me discuss the most important ways it is impacted by rewrite rules.

4.6.1 Blocking tags

The constraint solver needs to know when a reduction is blocked on a particular metavariable. Usually it is possible to point out a single metavariable, but this is no longer the case when rewrite rules are involved:

- With overlapping rewrite rules, reduction can be blocked on a set of metavariables. For example, if we try to reduce the expression $X + Y$ where X and Y are metavariables of type `Nat` and `+_+` is defined with the rewrite rules from Sect. 2.1, then this expression might reduce further when either X or Y is instantiated to a constructor. So a postponed constraint involving this expression has to be woken up when either metavariable is instantiated.
- For higher-order matching, matching checks whether a particular variable occurs freely in the body of a lambda or pi. When metavariables are involved, a variable occurrence may be *flexible*: whether or not the variable occurs depends on the instantiation of a particular metavariable²⁷. In this case reduction is blocked on the set of all metavariables with potentially unbound variables in their arguments.
- When a *conditional* rewrite rule is blocked on the conversion check because of an unsolved metavariable, reduction can be blocked on the metavariable that is preventing the conversion check from succeeding.^{28,29}

Currently the Agda implementation uses only an approximation of the set of metavariables it encounters, i.e. only the first metavariable encountered. This is harmless because the current implementation of Agda will eventually try again to solve all postponed constraints. If in the future Agda would be changed to be more careful in when it decided to wake up postponed constraints, a more precise tracking of blocking metavariables would also be desirable.

²⁶ See also <https://github.com/agda/agda/issues/3971>.

²⁷ <https://github.com/agda/agda/issues/1663>

²⁸ <https://github.com/agda/agda/issues/1987>

²⁹ <https://github.com/agda/agda/issues/2302>

4.6.2 Pruning and constructor-like symbols

When adding new rewrite rules, we also keep track of what symbols are *constructor-like*. This is important for the pruning phase of the constraint solver. For example, let us consider a constraint $X \stackrel{?}{=} Y (f x)$. Since the metavariable X does not depend on the variable x , the constraint solver attempts to *prune* the dependency of Y on x . If f is a regular postulate without any rewrite rules, there is no way that Y could depend on $f x$ without also depending on x , so the dependency of Y on its first argument is pruned away. However, if there is a rewrite rule where f plays the role of a constructor – say a rule $g (f y) \rightarrow \text{true}$ – then the assignment $X := \text{true}$ and $Y := \lambda y. g y$ is a valid solution to the constraint where Y *does* depend on its argument, so it should **not** be pruned away. In general, an argument should not be pruned if the head symbol is constructor-like, i.e. if there is at least one rewrite rule that matches against the symbol.

5 Related work

The idea of extending dependent type theory with rewrite rules is not new and has been studied from many different angles. The groundwork of all this work was laid in 1988 by Breazu-Tannen [29], who extended simply typed lambda calculus with first-order and higher-order rewrite rules (but not higher-order matching). In what follows, I give an overview of some important milestones, focussing on languages that combine dependent types and higher-order rewrite rules.

The idea of extending dependent type theory with rewrite rules originates in the work by Barbanera, Fernandez, and Geuvers [5]. They present the *algebraic λ -cube*, an extension of the λ -cube with algebraic rewrite rules, and study conditions for strong normalization. Since the left-hand sides of rewrite rules must be algebraic terms, this work does not include higher-order matching.

Several lines of work investigate possible ways to integrate rewrite rules into the Calculus of Constructions, with or without inductive datatypes:

- Walukiewicz-Chrząszcz [31] extends the calculus of constructions with inductive types and rewrite rules, and gives a termination criterion based on HORPO (higher-order recursive path ordering). Later, Walukiewicz-Chrząszcz and Chrząszcz also discuss the question of completeness and consistency of this system [32], and consider the addition of rewrite rules to the Coq proof assistant [12].
- The Open Calculus of Constructions [26, 27] integrates features from the Calculus of Constructions (CoC) with conditional rewrite rules, as well as other kinds of equational reasoning. It provides many of the same benefits as our system and is even more powerful when it comes to conditional rewrite rules. However it again does not provide higher-order matching or η -equality. It has a prototype implementation using the Maude language [13].
- The Calculus of Algebraic Constructions (CAC) [7] is another extension of the Calculus of Constructions with functions and predicates defined by higher-order rewrite rules. Compared to our implementation of rewrite rules, CAC is more limited in that it does not allow higher-order matching, but it provides criteria for checking subject reduction and strong normalization of the rewrite rules.

Coq modulo theory (CoqMT) [28] and the newer version CoqMTU [6, 18] extend the Coq proof assistant with a decidable theory. The equational theory in CoqMTU must be first-order, but can include equational rules such as commutativity, which cannot be expressed

as rewrite rules. CoqMTU also provides strong guarantees for confluence, subject reduction, strong normalization, and consistency of the theory. Unfortunately, the implementation of CoqMTU³⁰ has not been updated to work with the current version of Coq.

Our extension of dependent type theory with rewrite rules resembles in many ways the Dedukti system [15, 24, 10, 4]. Both systems support dependent types and higher-order non-linear rewrite rules. There are however some important differences:

- Dedukti was built up from the ground based on rewrite rules. In contrast, we start from a general dependently typed language (Agda) and extend it with rewrite rules.
- Dedukti is based on the Logical Framework (LF) [17], while our language is build from Martin-Löf's intuitionistic type theory [19], which includes several features not present in LF such as sigma types, W-types, identity types, and a universe hierarchy.
- Dedukti has universes à la Tarski: a universe is a set of codes that can be interpreted as types by an interpretation function. In contrast, Agda uses universes à la Russell: elements of a universe *are* types without need of an interpretation function.
- Dedukti uses an untyped conversion algorithm, while Agda uses a typed one. Hence we can support η -equality for functions and record types, which is not possible (directly) in Dedukti.
- Dedukti provides external tools for checking confluence and termination of the rewrite system given by the user. Applying the same strategy to rewrite rules in Agda would be difficult because several features cannot be translated into standard rewrite systems, e.g. copattern matching, eta-equality, irrelevance, and universe levels. All of these features introduce additional definitional equalities that should be taken into account when computing critical pairs. A confluence checker that does not would not detect all critical pairs and thus only be of limited use. Instead, we are currently working on integrating a confluence checker into Agda directly.³¹

The Zombie language [25] is another dependently typed language where definitional equality can be extended with user-provided equations that are applied automatically by the typechecker. Instead of rewrite rules, *Zombie* computes the congruence closure of the given equations and uses this during conversion checking. An important difference with our approach is that the definitional equality in *Zombie* does not include β -equality, which makes it easier to extend it in other directions. The congruence closure algorithm used by *Zombie* is untyped, which means it cannot handle η -equality of functions or records. It also does not include higher-order matching.

Our treatment of rewrite rules in parametrized modules is very similar to the one given by Chrzęszcz [11]. The main difference is that Chrzęszcz considers modules parametrized by other modules, while in Agda modules are parametrized by term variables. So our system is a bit simpler since we cannot have rewrite rules as parameters.

6 Future work

Safe(r) rewrite rules

This paper is about how to add rewrite rules to Agda or similar languages. By their design rewrite rules are a very unsafe feature of Agda. Compared to using `postulate`, rewrite rules by themselves do not break logical soundness of the theory, since it can only be used to turn

³⁰ <https://github.com/strub/coqmt>

³¹ The development version of Agda already includes an experimental flag `--confluence-check`, checking *local* confluence of rewrite rules.

propositional equalities into definitional ones. Logical consistency of the system thus follows from the consistency of extensional type theory. While several of our examples do introduce equalities that were previously unprovable (specifically higher inductive types, quotient types, exceptions, and observational equality), they do so by first explicitly postulating the required propositional equality and then registering it as a rewrite rule. Hence as long as the postulates preserve logical soundness, we can trust that turning them into rewrite rules does not break soundness either.

On the other hand, rewrite rules can break core assumptions of Agda such as confluence of reduction and even type preservation. So using rewrite rules is like building your own type theory, which means you have to do your own meta-theory to make sure everything is safe. Ideally, Agda would be able to detect if a given set of rewrite rules is “safe”, in the sense that they do not break the usual properties of Agda programs such as subject reduction and decidable typechecking. The development version of Agda 2.6.1 includes an experimental flag `--confluence-check`, which checks the *local* confluence of the declared rewrite rules. We are currently working to improve this confluence checker to also enforce *global* confluence of the rewrite rules. This would allow us to prove injectivity of Π types, and hence subject reduction of our type theory. For checking termination – and hence decidability of typechecking – we could make use of the dependency pairs criterion as done by `SizeChangeTool` for `Dedukti` [9].

Local rewrite rules

When programming in a dependently typed language, we rely on terms computing to their values. However, this fails when we work with abstract values (e.g. module parameters): until they are instantiated, they are opaque symbols without any computational behaviour. This actively encourages users to work with concrete values and discourages abstraction.

To improve this situation, we could allow *local* rewrite rules on module parameters to be added to the context. For example, we could parametrize a module over a value \emptyset and a binary operation `__·__` together with rewrite rules $\emptyset \cdot y \longrightarrow y$ and $x \cdot \emptyset \longrightarrow x$. When instantiating the module parameters, we have to check that the given instantiation of the parameters satisfies each of the rewrite rules as a definitional equality.

Having local rewrite rules greatly complicates checking of confluence and termination. So the future will have to point out if there is a reasonable way to allow local rewrite rules while maintaining subject reduction of the language.

Custom η rules

Rewrite rules allow us to add custom β rules to our type theory, but it would be useful to also allow custom η rules. This would for example allow us to add η -rules for datatypes such as `Vec`, making any vector of length `zero` definitionally equal to `[]`.

Where rewrite rules allow extending the reduction relation of the theory, custom η rules would allow extending the conversion checker directly. Since conversion in Agda is type-directed, it would make sense to allow custom η rules that match against the type of a constraint. Thus much of the matching algorithm in this paper could be reused for η rules.

7 Conclusion

This paper documents the process of integrating user-defined rewrite rules into a general-purpose dependently typed language, and all the weird interactions that I encountered along the way. Rewrite rules allow you to extend the power of a dependently typed language on a

much deeper level than normally allowed. They can be used as a convenient feature to make more terms typecheck without using explicit `rewrite` statements, and they allow advanced users to experiment with new evaluation rules, without actually modifying the typechecker. If you are an Agda user, I hope reading this paper has given you a deeper understanding of rewrite rules and allows you to harness their power responsibly. And if you are implementing your own dependently typed language, I hope you consider adding rewrite rules as a way to make it both easier to use and more extensible.

References

- 1 Agda development team. *Agda 2.6.1 documentation*, 2020. URL: <http://agda.readthedocs.io/en/v2.6.1/>.
- 2 Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In Stephanie Weirich, editor, *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*, pages 13–24. ACM, 2013. doi:10.1145/2502409.2502411.
- 3 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68. ACM, 2007.
- 4 Ali Assaf and Guillaume Burel. Translating HOL to dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015.*, volume 186 of *EPTCS*, pages 74–88, 2015. doi:10.4204/EPTCS.186.8.
- 5 Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-lambda-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.
- 6 Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub, and Qian Wang. CoQMTU: A higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 143–151. IEEE Computer Society, 2011. doi:10.1109/LICS.2011.37.
- 7 Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005. doi:10.1017/S0960129504004426.
- 8 Frédéric Blanqui. Type safety of rewriting rules in dependent types. In *At the 26th International Conference on Types for Proofs and Programs (TYPES 2020)*, 2020.
- 9 Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany.*, volume 131 of *LIPICs*, pages 9:1–9:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPICs.FSCD.2019.9.
- 10 Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The $\lambda\pi$ -calculus modulo as a universal proof language. In *the Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012)*, 2012.
- 11 Jacek Chrzaszcz. Modules in Coq are and will be correct. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2003. URL: <http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3085&page=130>.

- 12 Jacek Chrzyszcz and Daria Walukiewicz-Chrzyszcz. Towards rewriting in Coq. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 113–131. Springer, 2007. doi:10.1007/978-3-540-73147-4_6.
- 13 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. doi:10.1016/S0304-3975(01)00359-0.
- 14 Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns - definitional equality for all. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2014. doi:10.1007/978-3-642-54833-8_6.
- 15 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007. doi:10.1007/978-3-540-73228-0_9.
- 16 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *PACMPL*, 3(POPL):3:1–3:28, 2019. doi:10.1145/3290316.
- 17 Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060.
- 18 Jean-Pierre Jouannaud and Pierre-Yves Strub. Coq without type casts: A complete proof of Coq modulo theory. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 474–489. EasyChair, 2017. URL: <http://www.easychair.org/publications/paper/340342>.
- 19 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.
- 20 Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, 1998. doi:10.1016/S0304-3975(97)00143-6.
- 21 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991. doi:10.1093/logcom/1.4.497.
- 22 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- 23 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option - an exceptional type theory. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, *Lecture Notes in Computer Science*, pages 245–271. Springer, 2018. doi:10.1007/978-3-319-89884-1_9.
- 24 Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, MINES ParisTech, 2015.
- 25 Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 369–382. ACM, 2015. doi:10.1145/2676726.2676974.
- 26 Mark-Oliver Stehr. The open calculus of constructions (part i): An equational type theory with dependent types for programming, specification, and interactive theorem proving. *Fundamenta Informaticae*, 68(1-2):131–174, 2005.

- 27 Mark-Oliver Stehr. The open calculus of constructions (part ii): An equational type theory with dependent types for programming, specification, and interactive theorem proving. *Fundamenta Informaticae*, 68(3):249–288, 2005.
- 28 Pierre-Yves Strub. Coq Modulo Theory. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2010. doi:10.1007/978-3-642-15205-4_40.
- 29 Val Tannen. Combining algebra and higher-order types. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*, pages 82–90. IEEE Computer Society, 1988.
- 30 The Univalent Foundations Program. *Homotopy Type Theory - Univalent Foundations of Mathematics: The Univalent Foundations Program*. Institute for Advanced Study, 2013. URL: <https://homotopytypetheory.org/book/>.
- 31 Daria Walukiewicz-Chrzaszcz. Termination of rewriting in the calculus of constructions. *Journal of Functional Programming*, 13(2):339–414, 2003. doi:10.1017/S0956796802004641.
- 32 Daria Walukiewicz-Chrzaszcz and Jacek Chrzaszcz. Consistency and completeness of rewriting in the calculus of constructions. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 619–631. Springer, 2006. doi:10.1007/11814771_50.

A Complete rules of type theory with user-defined rewrite rules

A.1 Syntax

Terms. $\boxed{u, v, w, A, B, C, p, q}$

u, v, w, A, B, C, p, q	$::=$	$x \bar{u}$	(variable applied to zero or more arguments)
		$f \bar{u}$	(function symbol applied to zero or more arguments)
		$\lambda x. u$	(lambda abstraction)
		$(x : A) \rightarrow B$	(dependent function type)
		Set_i	(i th universe)

Substitutions. Substitutions $\boxed{\sigma}$ are lists of variable-term pairs $[u_1 / x_1, \dots, u_n / x_n]$. Application of a substitution to a term $\boxed{u\sigma}$ is defined as usual, avoiding variable capture by α -renaming where necessary.

Application. Application $\boxed{u v}$ is a partial operation on terms and is defined as follows:

$$\begin{aligned} (x \bar{u}) v &= x (\bar{u}; v) \\ (f \bar{u}) v &= f (\bar{u}; v) \\ (\lambda x. u) v &= u[v / x] \end{aligned}$$

Contexts. $\boxed{\Gamma, \Delta, \Phi, \Xi}$

$$\begin{aligned} \Gamma, \Delta, \Phi, \Xi &::= \cdot && \text{(empty context)} \\ &| \Gamma(x : A) && \text{(context extension)} \end{aligned}$$

Declarations. \boxed{d}

$$d ::= f : A \quad (\text{function symbol}) \\ | \quad \forall \Delta. f \bar{p} : A \longrightarrow v \quad (\text{rewrite rule})$$

A.2 Typing rules

We assume a global signature Σ containing declarations and rewrite rules, which is implicit in all the judgements.

Typing. $\boxed{\Gamma \vdash u : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad \frac{f : A \in \Sigma}{\Gamma \vdash f : A} \quad \frac{\Gamma(x : A) \vdash u : B}{\Gamma \vdash \lambda x. u : (x : A) \rightarrow B} \quad \frac{\Gamma \vdash u : (x : A) \rightarrow B \quad \Gamma \vdash v : A}{\Gamma \vdash u v : B[v/x]}$$

$$\frac{\Gamma \vdash A : \text{Set}_i \quad \Gamma(x : A) \vdash B : \text{Set}_j}{\Gamma \vdash (x : A) \rightarrow B : \text{Set}_{i \sqcup j}} \quad \frac{}{\text{Set}_i : \text{Set}_{1+i}} \quad \frac{\Gamma \vdash A = B : \text{Set}_i \quad \Gamma \vdash u : A}{\Gamma \vdash u : B}$$

Conversion. $\boxed{\Gamma \vdash u = v : A}$

$$\frac{\Gamma \vdash u \longrightarrow u' \quad \Gamma \vdash u' = v : A}{\Gamma \vdash u = v : A} \quad \frac{\Gamma \vdash v \longrightarrow v' \quad \Gamma \vdash u = v' : A}{\Gamma \vdash u = v : A} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x = x : A}$$

$$\frac{f : A \in \Sigma}{\Gamma \vdash f = f : A} \quad \frac{\Gamma(x : A) \vdash u x = v x : B}{\Gamma \vdash u = v : (x : A) \rightarrow B} \quad \frac{\Gamma \vdash u_1 = u_2 : (x : A) \rightarrow B \quad \Gamma \vdash v_1 = v_2 : A}{\Gamma \vdash u_1 v_1 = u_2 v_2 : B[v_1/x]}$$

$$\frac{\Gamma \vdash A_1 = A_2 : \text{Set}_i \quad \Gamma(x : A_1) \vdash B_1 = B_2 : \text{Set}_j}{\Gamma \vdash (x : A_1) \rightarrow B_1 = (x : A_2) \rightarrow B_2 : \text{Set}_{i \sqcup j}}$$

Reduction. $\boxed{\Gamma \vdash u \longrightarrow v}$

$$\frac{f : B \in \Sigma \quad (\forall \Xi. f \bar{p} : C \longrightarrow v) \in \Sigma \quad \Gamma \Xi; \cdot \vdash [(\bullet : B) \bar{u} // \bar{p}] \Rightarrow \sigma; \Psi \quad \forall (\Phi \vdash v \stackrel{?}{=} p : A) \in \Psi. \Gamma \Phi \vdash v = p \sigma : A}{\Gamma \vdash f \bar{u} \longrightarrow v \sigma}$$

Matching. $\boxed{\Gamma; \Phi \vdash [u : A // p] \Rightarrow \sigma; \Psi}$

$$\frac{x : B \in \Gamma}{\Gamma; \Phi \vdash [u : A // x] \Rightarrow [u/x]; \emptyset} \quad \frac{}{\Gamma; \Phi \vdash [u : A // v] \Rightarrow []; \{\Phi \vdash u \stackrel{?}{=} v : A\}}$$

$$\frac{\Gamma \Phi \vdash u \longrightarrow^* f \bar{v} \quad f : B \in \Sigma \quad \Gamma; \Phi \vdash [(\bullet : B) \bar{v} // \bar{p}] \Rightarrow \sigma; \Psi}{\Gamma; \Phi \vdash [u : A // f \bar{p}] \Rightarrow \sigma; \Psi}$$

$$\frac{\Gamma \Phi \vdash u \longrightarrow^* x \bar{v} \quad x : B \in \Phi \quad \Gamma; \Phi \vdash [(\bullet : B) \bar{v} // \bar{p}] \Rightarrow \sigma; \Psi}{\Gamma; \Phi \vdash [u : A // x \bar{p}] \Rightarrow \sigma; \Psi}$$

$$\frac{\Gamma \Phi \vdash A \longrightarrow^* (x : B) \rightarrow C \quad \Gamma; \Phi(x : B) \vdash [u x : C // p x] \Rightarrow \sigma; \Psi}{\Gamma; \Phi \vdash [u : A // p] \Rightarrow \sigma; \Psi}$$

$$\frac{\Gamma \Phi \vdash A \longrightarrow^* (x : B) \rightarrow C \quad \Gamma; \Phi \vdash [B // p] \Rightarrow \sigma_1; \Psi_1 \quad \Gamma; \Phi(x : B) \vdash [C // q] \Rightarrow \sigma_2; \Psi_2}{\Gamma; \Phi \vdash [A : D // (x : p) \rightarrow q] \Rightarrow \sigma_1 \uplus \sigma_2; \Psi_1 \cup \Psi_2}$$

Spine matching. $\boxed{\Gamma; \Phi \vdash [(\bullet : A) \bar{u} // \bar{p}] \Rightarrow \sigma; \Psi}$

$$\frac{\Gamma\Phi \vdash A \longrightarrow^* (x : B) \rightarrow C \quad \Gamma; \Phi \vdash [u : B // p] \Rightarrow \sigma_1; \Psi_1 \quad \Gamma; \Phi \vdash [(\bullet : C[u/x]) \bar{u} // \bar{p}] \Rightarrow \sigma_2; \Psi_2}{\Gamma; \Phi \vdash [(\bullet : A) \cdot // \cdot] \Rightarrow []; \emptyset} \quad \Gamma; \Phi \vdash [(\bullet : A) u; \bar{u} // p; \bar{p}] \Rightarrow \sigma_1 \uplus \sigma_2; \Psi_1 \cup \Psi_2$$

A.3 Checking declarations

A declaration of a function symbol $f : A$ is valid if $\Gamma \vdash A : \text{Set}_i$. A declaration of a rewrite rule $\forall \Delta. f \bar{p} : A \longrightarrow v$ is valid if:

- Each variable in Δ occurs at least once in a pattern position in \bar{p} .
- $\Delta \vdash f \bar{p} : A$ and $\Delta \vdash v : A$
- There is no term w such that $\Delta \vdash f \bar{p} \longrightarrow w$.

A Quantitative Understanding of Pattern Matching

Sandra Alves

DCC-FCUP & CRACS, University of Porto, Portugal

Delia Kesner

Université de Paris, CNRS, IRIF, France

Institut Universitaire de France, France

Daniel Ventura

INF, Universidade Federal de Goiás, Goiânia, Brazil

Abstract

This paper shows that the recent approach to quantitative typing systems for programming languages can be extended to pattern matching features. Indeed, we define two resource-aware type systems, named \mathcal{U} and \mathcal{E} , for a λ -calculus equipped with pairs for both patterns and terms. Our typing systems borrow some basic ideas from [19], which characterises (head) normalisation in a *qualitative* way, in the sense that typability and normalisation coincide. But, in contrast to [19], our systems also provide *quantitative* information about the *dynamics* of the calculus. Indeed, system \mathcal{U} provides *upper bounds* for the *length* of (head) normalisation sequences *plus* the *size* of their corresponding normal forms, while system \mathcal{E} , which can be seen as a refinement of system \mathcal{U} , produces *exact bounds* for *each* of them. This is achieved by means of a non-idempotent intersection type system equipped with different technical tools. First of all, we use product types to type pairs instead of the disjoint unions in [19], which turn out to be an essential quantitative tool because they remove the confusion between “being a pair” and “being duplicable”. Secondly, typing sequents in system \mathcal{E} are decorated with tuples of integers, which provide quantitative information about normalisation sequences, notably *time* (cf. length) and *space* (cf. size). Moreover, the time resource information is remarkably refined, because it discriminates between different kinds of reduction steps performed during evaluation, so that beta, substitution and matching steps are counted separately. Another key tool of system \mathcal{E} is that the type system distinguishes between *consuming* (contributing to time) and *persistent* (contributing to space) constructors.

2012 ACM Subject Classification Theory of computation \rightarrow Lambda calculus; Theory of computation \rightarrow Operational semantics

Keywords and phrases Intersection Types, Pattern Matching, Exact Bounds

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.3

Funding This work was partially done within the framework of ANR COCA HOLA (16-CE40-004-01). It was partially funded by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020 and by the Brazilian Research Council (CNPq) grant Universal 430667/2016-7.

Acknowledgements We are grateful to Antonio Bucciarelli and Simona Ronchi Della Rocca for fruitful discussions.

1 Introduction

Pattern matching mechanisms are used in several modern programming languages and proof assistants as they provide an efficient way to process and decompose data. However, the semantics of programming languages usually focus on λ -calculi –a much more basic formalism– thus causing a conceptual gap between theory and practice, simply because some properties of the λ -calculus do not translate directly to languages with matching primitives. Several



© Sandra Alves, Delia Kesner, and Daniel Ventura;
licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 3; pp. 3:1–3:36

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

examples of this mismatch can be cited, e.g. solvability [19], standardisation for pattern calculi [37], and neededness [15]. It is then crucial to study the semantics of programming languages with pattern matching features by means of formal calculi equipped with built-in patterns, referred to as *pattern calculi* (e.g. [21, 35, 32, 37, 6]).

The notion of λ -abstraction in pattern-calculi is generalised to functions of the form $\lambda p.t$, where p is a *pattern* specifying the expected structure of their arguments. For instance, in calculi equipped with *pair constructors* for both patterns and terms, the term $\lambda\langle x, y \rangle.x$ becomes a valid abstraction, to be only successfully evaluated against pairs, i.e. arguments of the form $\langle u, v \rangle$, and yielding the first projection of this pair, i.e. the first component u of the pair. In this work we focus on such a pattern calculus. This can be seen as a simplified form of *algebraic* pattern matching, but still powerful enough to reason about the most interesting features of existing syntactical matching mechanisms.

Type information, and in particular the size of arbitrary type derivations in some special type disciplines, has been used as a powerful quantitative tool to reason about *time* (*length* of evaluation sequences) and *space* (*size* of normal forms). More precisely, when t evaluates to t' , then the size of the type derivation of t' is smaller than that of t , thus the size of type derivations provides an *upper bound* for the length of normalisation sequences as well as for the size of their corresponding normal forms. This was first done for the (call-by-name) notions of head and leftmost evaluation implemented by two variants of the Krivine's abstract machine (KAM) [22, 23, 44], and it was later appropriately extended to other formalisms, e.g. [8, 20, 25, 38].

Now we discuss some interesting features of the underlying type system that we use in this paper. While (idempotent) *intersection types* [11] allow terms to be typed with distinct types by means of an intersection operator \cap , which verifies not only associativity and commutativity but also idempotency given by $\sigma \cap \sigma = \sigma$, *non-idempotent* intersection types distinguishes between $\sigma \cap \sigma$ and σ , thus also discriminating *quantitative* information in type derivations. For this reason, idempotent (resp. non-idempotent) types are often represented by sets (resp. *multisets*). For example, the term $\lambda x.\lambda y.xyxy$ can be typed with $\{\{\sigma\} \rightarrow \{\sigma\} \rightarrow \tau\} \rightarrow \{\sigma\} \rightarrow \tau$ in the first model, while the non-idempotent version becomes $[[\sigma] \rightarrow [\sigma] \rightarrow \tau] \rightarrow [\sigma, \sigma] \rightarrow \tau$. As a consequence, a type derivation for $(\lambda x.\lambda y.xyxy)uv : \tau$ in the idempotent system only depends on one type derivation for $u : \{\sigma\} \rightarrow \{\sigma\} \rightarrow \tau$ and another one for $v : \sigma$, while for its reduct $uvv : \tau$, two derivations for $v : \sigma$ are needed. In contrast, a type derivation in the non-idempotent system already requires two derivations for $v : \sigma$ to correctly infer $(\lambda x.\lambda y.xyxy)uv : \tau$. Therefore, while type derivations may increase after reductions in the former, they decrease in the latter.

Non-idempotent intersection (also called nowadays *quantitative*) type systems, have been independently introduced in the framework of the λ -calculus by Gardner [27] and Kfoury [43]. Although widely unnoticed, the *quantitative* power of such systems turned out to be crucial in several resource aware consumption investigations. It was only after [16] that this quantitative feature was highlighted, and since De Carvalho's thesis in 2007 (see also [22]) its relation with linear logic [28] and quantitative relational models has been deeply explored. As its idempotent counterpart, non-idempotent intersection type systems may characterise different notions of normalisation (such as head, weak and strong) [23, 13, 20] but, instead of using some semantical argument (e.g. reducibility) to prove such a characterisation, simple combinatorial arguments are enough to guarantee normalisation of typable terms.

If instead of upper bounds one wants to obtain *exact bounds*, then the crucial point is to measure only *minimal* typing derivations, which give the notion of *all and only information* for typings (cf. [56] for an abstract definition). Syntactic notions of minimal typings were

supplied for the *head evaluation* strategy implemented by the KAM [22], then for the *maximal evaluation* strategy [13] for the λ -calculus. The technique was further developed in [2] with the introduction of an appropriate notion of *tightness* capturing minimal typings, thus systematically broadening the definition of exact bounds for different evaluation strategy of the λ -calculus. In all these works, it is possible to extract from a (minimal) type derivation, both the length of the reduction sequence to normal form as well as the size of this normal form. The tightness technique was also applied for call-by-value [3], call-by-need [4], linear head evaluation [3], as well as for several evaluation strategies in classical logic [42]. Our paper extends these results to a λ -calculus with pair pattern matching by providing two sound and complete typing systems, named \mathcal{U} and \mathcal{E} , that respectively provide upper bounds an exact measures for the length of (head) normalisation sequences, as well as for the size of the corresponding reached normal forms.

Contributions

The first contribution of this paper is to go beyond the *qualitative* characterisation of head normalisation for pair pattern calculi [19] by providing a typing system \mathcal{U} being able to compute *upper bounds* for head evaluation. To achieve this, we have introduced different key tools on the untyped side –the reduction calculus– as well as on the typed side –the type system itself.

On the *untyped* side, one of the main reasons why the type system in [19] fails to provide upper bounds or exact measures for head normalisation is because *commuting conversions* are considered as *independent* rules of the reduction relation associated to the underlying pattern calculus. A typical example is the commuting rule $t[p\backslash v]u \mapsto_{\sigma} (tu)[p\backslash v]$ pushing out an explicit matching from an application when there is no capture of free variables. Indeed, the size of type derivations is not strictly decreasing w.r.t. commuting conversions. We solve this problem by integrating these (structural) commuting conversions into the non-structural operational reduction rules, so that the resulting system, based on *explicit matchings*, implements reduction *at a distance* [5]. Thus for example, the operational Beta-rule $(\lambda p.t)u \mapsto t[p\backslash u]$ in [19] becomes here $L[\lambda p.t]u \mapsto L[t[p\backslash u]]$, which combines the commuting conversion \mapsto_{σ} with the (non-structural) Beta-reduction rule into a single rule. Even more interesting cases are presented in Sec. 2. Moreover, our presentation provides a suitable *deterministic* head evaluation strategy which is complete w.r.t. the (non-deterministic) notion of head-normalisation defined in [19], in the sense that both notions turn out to be equivalent, thus answering one of the open questions in [19].

On the *typed* side, we adopt standard product types specified by means of a *pair type*. This stands in contrast to the disjoint unions used in [19], which have an important undesirable consequence, because multisets of types in this model carry two completely different meanings: being a pair (but not necessarily a duplicable pair), or being a duplicable term (but not necessarily a pair). Our product types restore a crucial idea in non-idempotent type theory: multisets of types are only assigned to terms that are going to be duplicated during evaluation.

The new specification of the deterministic reduction system at a distance is now well-behaved w.r.t. our first type system \mathcal{U} : if t is well typed in \mathcal{U} , then the size of its type derivation gives an upper bound to the (deterministic) head-reduction sequence from t to its (head) normal form. Our system \mathcal{U} can then be seen as a form of quantitative (relational) *model* for the pair pattern calculus (Sec. 4), following the lines of [17, 18, 48].

The second contribution of this paper is to go beyond upper bounds by providing a typing system \mathcal{E} being able to provide *exact bounds* for head evaluation. This is done by using several key tools.

An important notion used in system \mathcal{E} is the clear distinction between *consuming* and *persistent* constructors. This has some intuition coming from the theory of residuals [10], where any symbol occurring in a normal form can be traced back to the original term. A constructor is consuming (resp. persistent) if it is consumed (resp. not consumed) during head-reduction. For instance, in $(\lambda z.z)(\lambda z.z)$ the first abstraction is consuming while the second one is persistent. This dichotomy between consuming/persistent constructors has already been highlighted in [41, 42] for the $\lambda\mu$ -calculus, and it is adapted here for the pattern calculus. Indeed, patterns and terms are consumed when the pair constructor is destroyed during the execution of the pattern matching rule. Otherwise, patterns and pairs are persistent, and they do appear in the normal form of the original term. For example, in the term $(\lambda z.(\lambda\langle x, y \rangle.\mathbb{I}zz)\langle u, v \rangle)$, the pair $\langle u, v \rangle$ is going to be duplicated, but only one of its copies is going to be consumed by the matching operations. The other copy will be persistent and contribute to the normal form of the term.

Another major ingredient of our approach is the use of *tight* types, inspired by [2], and the corresponding notion of tight (cf. minimal) derivations. This is combined with the introduction of *counters* in the typing judgements, which are now of the form $\Gamma \vdash^{(b,e,m,f)} t : \sigma$. These counters are used to discriminate between the different sorts of reduction steps performed during evaluation, so that firing beta (b), computing substitution (e) or matching (m) steps are exactly and independently counted for each tight type derivation. More precisely, *soundness* of our system \mathcal{E} guarantees that if a judgement $\Gamma \vdash^{(b,e,m,f)} t : \sigma$ is tightly derivable, then b (resp e and m) corresponds to the number of beta firing (resp. substitution and matching) rules used to head evaluate the term t , while f is exactly the size of the corresponding normal form. Moreover, *completeness*, given by the reverse implication of the previous statement, also holds.

The following list summarises our contributions:

- A deterministic head-strategy for the pattern calculus which is complete w.r.t. the notion of head-normalisation.
- A sound and complete type system \mathcal{U} , which provides upper bounds for the length of head-normalisation sequences plus the size of its corresponding normal forms.
- Refinement of system \mathcal{U} to a sound and complete system \mathcal{E} , being able to provide independent exact bounds for both the length of head-normalisation sequences and the size of its corresponding normal forms.

Other Related Works

Non-idempotent intersection types have been applied to the λ -calculus for the characterisation of termination with respect to a variety of evaluation strategies, such as call-by-value [25, 3], call-by-need [36, 8, 4] and (linear) head reduction [27, 2]. They have been well-adapted also to some explicit resource calculi [13, 38, 39], as well as to pattern calculi [12, 19, 9], proof-nets [24], classical logic [40, 42] and call-by-push-value [26, 29].

Closer to our work, non-idempotent intersection types have been used to characterise strong normalisation in a calculus with fix-point operators and pattern matching on constructors [12]. Similarly, a strong call-by-need strategy for a pattern matching language was defined in [9], and completeness of the strategy was shown by means of non-idempotent intersection types by extending the technique introduced in [36, 8]. In both cases, despite the use of non-idempotent types, the result was qualitative, as no quantitative results were obtained by means of the typing system.

Even closer to our work, [19] studied the solvability property in a pair pattern calculus, the main result being that solvability is equivalent to typability plus inhabitation in a non-idempotent intersection type system. One of the contributions of [19] is a characterisation of

(non-deterministic) head-normalisation by means of typability, which is merely *qualitative*, as it does not give any *upper bound/exact measure* for head-evaluation, as discussed in the previous subsection.

More practical type-based approaches (i.e. mostly sound but not necessarily complete) to quantitative analysis are sized-types [49, 54, 7, 45, 46] and (automatic) amortised resource analysis [33, 53, 30, 34, 31, 51].

Sized Types. This line of work is based on the use of types indexed by sizes, which are essentially ordinals. The approach is based on the fact that the compiler checks if the program is typed with the correct size, so that resource usages of programs can be derived from the sized types informations.

While space cost is determined in [54], upper bounds for both space and time costs are obtained in [49, 7]. In particular, sized types are used in [7] to obtain space bounds, while time bounds are computed by using a kind of clock, achieved by means of ticking monadic transformations, originally introduced in [55] as ticking monads to get time complexity for lazy languages. This is done for a call-by-value functional language with (a fixed set of) inductive datatypes, enriched with index polymorphism: functions can be polymorphic in their size annotations. In this respect, sized types enriched with intersection types have been used in [52] to handle time costs for a call-by-value strategy in a more restricted language. Sized types are also extended in [14] to guarantee termination of general higher-order rewriting. Yet another approach based on (first-order) size indices is given by linear dependent types [45, 46], where time and space bounds are obtained by establishing a relation with linear logic, a key tool used to define quantitative types through non-idempotent intersection types. Completeness depends on an oracle for the first order theory on indices describing the semantical properties of the function symbols, so the approach cannot be fully turned into an automatic tool.

Amortised resource analysis. This line of work is motivated by the fact that the worst-case run time analysis per operation, rather than per algorithm, can lead to very pessimistic complexity bounds. This is then replaced by an approach considering both the costly and less costly operations together over the whole set of operations of the algorithm. Automatisation of amortised resource analysis has been achieved in a series of works [33, 30, 53, 34, 31, 51], including space usage [53] and more general usages [31], all regarding a lazy functional language. The pioneer work in [30] has evolved to more sophisticated tools, leading today to RAML [50], a language applied to an industrial strength compiler [31]. Lazy evaluation is not handled in RAML, however, [51] proposes a practical tool to estimate resource usage for Haskell expressions.

Road-map: Sec. 2 introduces the pattern calculus. Sec. 3 presents the typing system \mathcal{U} , together with some of its quantitative properties, and Sec. 4 suggests a relational model for our pattern calculus based on the type system. In Sec. 5, we refine \mathcal{U} to extract exact bounds, which leads to the definition of our second typing system \mathcal{E} . The soundness (resp. completeness) proof for \mathcal{E} is given in Sec. 6 (resp. Sec. 7). Conclusions and future work are discussed in Sec. 8. All proofs are presented in the Appendix.

2 The Pattern Calculus

In this section we introduce the pattern calculus, an extension of the λ -calculus where abstraction is extended to *pair patterns* and terms are extended to *pairs*. We start by introducing the syntax of the calculus.

3:6 A Quantitative Understanding of Pattern Matching

Terms and contexts of the pattern calculus are defined by means of the following grammars:

(Patterns)	p, q	$::=$	$x \mid \langle p, q \rangle$
(Terms)	t, u, v	$::=$	$x \mid \lambda p.t \mid \langle t, u \rangle \mid tu \mid t[p \setminus u]$
(List Contexts)	L	$::=$	$\square \mid L[p \setminus u]$
(Contexts)	C	$::=$	$\square \mid \lambda p.C \mid \langle C, t \rangle \mid \langle t, C \rangle \mid Ct \mid tC \mid C[p \setminus t] \mid t[p \setminus C]$

where $x, y, z, w \dots$ range over a countable set of variables, and every pattern p is assumed to be *linear*, i.e. every variable appears at most once in p . The term x is called a **variable**, $\lambda p.t$ is an **abstraction**, $\langle t, u \rangle$ is a **pair**, tu is an **application** and $t[p \setminus u]$ is a **closure**, where $[p \setminus u]$ is an **explicit matching** operator. Special terms are $I := \lambda z.z$, $\Delta := \lambda z.zz$ and $\Omega := \Delta\Delta$. As usual we use the abbreviation $\lambda p_1 \dots p_n.t_1 \dots t_m$ for $\lambda p_1(\dots(\lambda p_n((t_1 t_2) \dots t_m)) \dots)$, $n \geq 0$, $m \geq 1$.

We write $\text{var}(p)$ to denote the variables in the pattern p . **Free and bound variables** of terms and contexts are defined as expected, in particular $\text{fv}(\lambda p.t) := \text{fv}(t) \setminus \text{var}(p)$, $\text{fv}(t[p \setminus u]) := (\text{fv}(t) \setminus \text{var}(p)) \cup \text{fv}(u)$ and $\text{bv}(\lambda p.t) := \text{bv}(t) \cup \text{var}(p)$, $\text{bv}(t[p \setminus u]) := \text{bv}(t) \cup \text{var}(p) \cup \text{bv}(u)$. We also define the **domain of a list context** as $\text{dlc}(\square) = \emptyset$ and $\text{dlc}(L[p \setminus u]) = \text{dlc}(L) \cup \text{var}(p)$. We write $p \# q$ if $\text{var}(p)$ and $\text{var}(q)$ are disjoint. As usual, terms are considered modulo α -conversion, so that for example $\lambda \langle x, y \rangle.xz =_\alpha \lambda \langle x', y' \rangle.x'z$ and $x[\langle x, y \rangle \setminus z] =_\alpha x'[\langle x', y' \rangle \setminus z]$. Given a list context L and a term t , $L[t]$ denotes the term obtained by replacing the unique occurrence of \square in L by t , possibly allowing the capture of free variables of t . We use $t\{x \setminus u\}$ to denote the meta-level substitution operation which replaces all the free occurrences of x in t by the term u . As usual, this operation is performed modulo α -conversion so that capture of free variables is avoided. We use the predicate $\text{abs}(t)$ when t is of the form $L[\lambda p.u]$. The **reduction relation** \rightarrow_p on terms is given by the closure over *all* contexts of the following rewriting rules.

$L[\lambda p.t]u$	\mapsto	$L[t[p \setminus u]]$	$\text{dlc}(L) \cap \text{fv}(u) = \emptyset$
$t[\langle p_1, p_2 \rangle \setminus \langle u_1, u_2 \rangle]$	\mapsto	$L[t[p_1 \setminus u_1][p_2 \setminus u_2]]$	$\text{dlc}(L) \cap \text{fv}(t) = \emptyset$
$t[x \setminus u]$	\mapsto	$t\{x \setminus u\}$	

The reduction relation \rightarrow_p defined above is related to that in [19], called \rightarrow_{Λ_p} , in the following sense: \rightarrow_{Λ_p} contains two subsystem relations, one to deal with *clashes*, which are not handled by the reduction system in the present calculus since we consider typable terms only (cf. Lem. 6), and another one containing the following five rules:

$(\lambda p.t)u$	\mapsto	$t[p \setminus u]$	
$t[\langle p_1, p_2 \rangle \setminus \langle u_1, u_2 \rangle]$	\mapsto	$t[p_1 \setminus u_1][p_2 \setminus u_2]$	
$t[x \setminus u]$	\mapsto	$t\{x \setminus u\}$	
$t[p \setminus v]u$	\mapsto	$(tu)[p \setminus v]$	$\text{fv}(u) \cap \text{var}(p) = \emptyset$
$t[\langle p_1, p_2 \rangle \setminus u][q \setminus v]$	\mapsto	$t[\langle p_1, p_2 \rangle \setminus u][q \setminus v]$	$\text{fv}(t) \cap \text{var}(q) = \emptyset$

The two last rules can be seen as commuting conversions, which are integrated in the first (two) rules of our reduction system \rightarrow_p by using the *substitution at a distance paradigm* [5]. It is worth noticing that $t \rightarrow_p t'$ can be simulated by $t \rightarrow_{\Lambda_p}^+ t'$. For instance, $(\lambda p.t)[p_1 \setminus u_1][p_2 \setminus u_2]u \rightarrow_p t[p \setminus u][p_1 \setminus u_1][p_2 \setminus u_2]$ can be simulated by:

$$\begin{aligned}
 (\lambda p.t)[p_1 \setminus u_1][p_2 \setminus u_2]u &\rightarrow_{\Lambda_p} ((\lambda p.t)[p_1 \setminus u_1]u)[p_2 \setminus u_2] \rightarrow_{\Lambda_p} ((\lambda p.t)u)[p_1 \setminus u_1][p_2 \setminus u_2] \\
 &\rightarrow_{\Lambda_p} t[p \setminus u][p_1 \setminus u_1][p_2 \setminus u_2]
 \end{aligned}$$

Our formulation of the pattern calculus at a distance, given by the relation \rightarrow_p , as well as the corresponding head strategy that we present below, are one of the essential untyped tools used in this paper to get quantitative results about head-normalisation (cf. Sec. 3 and Sec. 5).

Although the reduction relation \rightarrow_p is *non-deterministic*, it can easily be shown to be *confluent*, for example using the same technique in [19]. However, in order to study exact bounds of evaluation, we need to define a *deterministic* strategy for the pattern calculus, i.e. a subrelation of \rightarrow_p that is able to compute the same normal forms. Fig. 1 gives an operational semantics for the pattern calculus, which turns out to be an extension of the well-known notion of *head-reduction* for λ -calculus, then also named **head-reduction**, and denoted by \rightarrow_h . In the following inductive definition $t \rightarrow_h u$ means that t head-reduces to u , and $t \not\rightarrow_h$ means that t is a **head normal-form**, i.e. there is no u such that $t \rightarrow_h u$.

$\frac{\text{dlc}(L) \cap \text{fv}(u) = \emptyset}{L[\lambda p.t]u \rightarrow_h L[t[p \setminus u]]} \text{ (b)}$	$\frac{t \not\rightarrow_h \quad \text{dlc}(L) \cap \text{fv}(t) = \emptyset}{t[\langle p_1, p_2 \rangle \setminus L[\langle u_1, u_2 \rangle]] \rightarrow_h L[t[p_1 \setminus u_1][p_2 \setminus u_2]]} \text{ (m)}$	$\frac{t \not\rightarrow_h}{t[x \setminus u] \rightarrow_h t\{x \setminus u\}} \text{ (e)}$	
$\frac{t \rightarrow_h t'}{\lambda p.t \rightarrow_h \lambda p.t'}$	$\frac{t \rightarrow_h t' \quad \text{-abs}(t)}{tu \rightarrow_h t'u}$	$\frac{t \rightarrow_h t'}{t[p \setminus u] \rightarrow_h t'[p \setminus u]}$	$\frac{t \not\rightarrow_h \quad p \neq x \quad u \rightarrow_h u'}{t[p \setminus u] \rightarrow_h t[p \setminus u']}$

■ **Figure 1** The head-reduction strategy for the pattern calculus.

Rule **b** fires the computation of terms by transforming an application of a function to an argument into a closure term. Decomposition of patterns and terms is performed by means of rule **m**, when a pair pattern is matched against a pair term. Substitution is performed by rule **e**, i.e. an explicit (simple) matching of the form $[x \setminus u]$ is executed. This form of syntactic pattern matching is very simple, and does not consider any kind of failure result, but is already expressive enough to specify the well-known mechanism of successful matching. Context closure is *similar* to the call-by-name λ -calculus case, but not exactly the same. Indeed, head-reduction is performed on the left-hand side of applications and closures whenever possible. Otherwise, arguments of explicit matching operators must be head-reduced in order to unblock these operators, i.e. in order to decompose $[p \setminus u]$ when p is a pair pattern but u is still not a pair. Notice however that when u is already a pair, no head-reduction inside u can take place, thus implementing a *lazy* strategy for pattern matching. Standardisation of calculi as the one in this paper has been studied in [37].

Given any (one-step) reduction relation $\rightarrow_{\mathcal{R}}$, we use $\rightarrow_{\mathcal{R}}^*$, or more precisely $\rightarrow_{\mathcal{R}}^k$ ($k \geq 0$) to denote the reflexive-transitive closure of $\rightarrow_{\mathcal{R}}$, i.e. the composition of k \mathcal{R} -steps. In the case of head-reduction, we may use the alternative notation $\rightarrow_h^{(b,e,m)}$ to emphasize the number of reduction steps in a given reduction sequence, i.e. if $\rho : t \rightarrow_h^{(b,e,m)} u$, then there are exactly b **b**-steps, e **e**-steps and m **m**-steps in the reduction sequence ρ . We will often use the notation \rightarrow_b to explicitly refer to a **b**-step (resp. \rightarrow_e and \rightarrow_m for **e** and **m** steps). The reduction relation \rightarrow_h is in fact a function:

► **Proposition 1.** *The relation \rightarrow_h is deterministic.*

► **Example 2.** Let us consider the combinators $\mathbf{I} := \lambda z.z$ and $\mathbf{K} := \lambda x_1.\lambda y_1.x_1$. Then we have $(\lambda\langle x, y \rangle.x(\mathbf{I}y))[z\backslash\mathbf{I}](\mathbf{I}(\mathbf{K}, w)) \rightarrow_{\mathbf{h}}^{(4,6,1)} \lambda y_1.w$:

$$\begin{array}{ll}
(\lambda\langle x, y \rangle.x(\mathbf{I}y))[z\backslash\mathbf{I}](\mathbf{I}(\mathbf{K}, w)) & \rightarrow_{\mathbf{b}} (x(\mathbf{I}y))[\langle x, y \rangle \backslash \mathbf{I}(\mathbf{K}, w)][z\backslash\mathbf{I}] \\
\rightarrow_{\mathbf{b}} (x(\mathbf{I}y))[\langle x, y \rangle \backslash z[z\backslash\langle \mathbf{K}, w \rangle]][z\backslash\mathbf{I}] & \rightarrow_{\mathbf{e}} (x(\mathbf{I}y))[\langle x, y \rangle \backslash \langle \mathbf{K}, w \rangle][z\backslash\mathbf{I}] \\
\rightarrow_{\mathbf{m}} (x(\mathbf{I}y))[x\backslash\mathbf{K}][y\backslash w][z\backslash\mathbf{I}] & \rightarrow_{\mathbf{e}} (\mathbf{K}(\mathbf{I}y))[y\backslash w][z\backslash\mathbf{I}] \\
\rightarrow_{\mathbf{b}} (\lambda y_1.x_1)[x_1\backslash\mathbf{I}y][y\backslash w][z\backslash\mathbf{I}] & \rightarrow_{\mathbf{e}} (\lambda y_1.\mathbf{I}y)[y\backslash w][z\backslash\mathbf{I}] \\
\rightarrow_{\mathbf{b}} (\lambda y_1.z[z\backslash y])[y\backslash w][z\backslash\mathbf{I}] & \rightarrow_{\mathbf{e}} (\lambda y_1.y)[y\backslash w][z\backslash\mathbf{I}] \\
\rightarrow_{\mathbf{e}} (\lambda y_1.w)[z\backslash\mathbf{I}] & \rightarrow_{\mathbf{e}} \lambda y_1.w
\end{array}$$

Head normal-forms may contain ill-formed terms called (*head*) clashes not representing a desired result for a computation, i.e. (head) terms not syntactically well-formed. For example, a pair applied to another term $\langle u_1, u_2 \rangle v$, or a matching between a pair pattern and a function $t[\langle p_1, p_2 \rangle \backslash \lambda p.u]$ are considered to be (head) normal clashes. Formally, a term is said to be a **(head) clash** if it is generated by the following grammar:

$$\begin{array}{ll}
\text{(Head Clash)} \quad \mathbf{U} & ::= c \mid \lambda p.\mathbf{U} \mid \mathbf{U}t \mid \mathbf{U}[p\backslash t] \mid t[\langle p_1, p_2 \rangle \backslash \mathbf{U}] \\
\text{(Clash)} \quad c & ::= \mathbf{L}[\langle u_1, u_2 \rangle]v \mid t[\langle p_1, p_2 \rangle \backslash \mathbf{L}[\lambda p.u]]
\end{array}$$

Then, a term t is said to be **(head) clash-free** if t does not head-reduce to a (head) clash, i.e. if there is no $u \in \mathbf{U}$ such that $t \rightarrow_{\mathbf{h}}^* u$. Remark in particular that every pair is (head) clash-free. A rewriting system raising a warning (i.e. a failure) when detecting a (head) clash has been defined in [19], allowing to restrict the attention to a smaller set of terms, called *canonical* terms, that are intended to be the (head) clash-free terms that are not reducible by the relation $\rightarrow_{\mathbf{h}}$. Canonical terms can be characterised inductively as follows:

$$\begin{array}{ll}
\text{(canonical forms)} \quad \mathcal{M} & ::= \lambda p.\mathcal{M} \mid \langle t, t \rangle \mid \mathcal{M}[\langle p_1, p_2 \rangle \backslash \mathcal{N}] \mid \mathcal{N} \\
\text{(pure canonical forms)} \quad \mathcal{N} & ::= x \mid \mathcal{N}t \mid \mathcal{N}[\langle p_1, p_2 \rangle \backslash \mathcal{N}]
\end{array}$$

In summary, canonical terms and irreducible terms are related as follows:

► **Proposition 3.** $t \in \mathcal{M}$ if and only if t is (head) clash-free and $t \not\rightarrow_{\mathbf{h}}$.

Size of canonical terms is given by: $|x| := 0$, $|\langle t, u \rangle| := 1$, $|\mathcal{N}t| := |\mathcal{N}| + 1$, $|\lambda p.\mathcal{M}| := |\mathcal{M}| + 1$, and $|\mathcal{M}[\langle p_1, p_2 \rangle \backslash \mathcal{N}]| := |\mathcal{M}| + |\mathcal{N}| + 1$. As an example, the terms $\lambda\langle x, y \rangle.\langle x, \mathbf{I} \rangle$ and $\lambda x.y(\langle x, z \rangle \mathbf{I})$ are canonical forms of size 2 while $x\Omega$ and $z[\langle z, w \rangle \backslash x\Omega]$ are pure canonical terms of size 1 and 2 respectively. The term $\langle x, \mathbf{I} \rangle w$ is none of them, and the term $\mathbf{I}x$ can head-reduce to the canonical term x .

Finally, we define a term t to be **head-normalisable** if there exists a canonical form $u \in \mathcal{M}$ such that $t \rightarrow_{\mathbf{p}}^* u$. Moreover, t is said to be **head-terminating** if there exists a canonical form $u \in \mathcal{M}$ and an integer $k \geq 0$ such that $t \rightarrow_{\mathbf{h}}^k u$. The relation between the non-deterministic reduction relation $\rightarrow_{\mathbf{p}}$ and the deterministic strategy $\rightarrow_{\mathbf{h}}$ will be established later, but we can already say that, while t head-terminating immediately implies t head-normalisable, the completeness of the head-strategy w.r.t. head-normalisation is not trivial (Thm. 7).

3 The \mathcal{U} Typing System

In this section we introduce our first typing system \mathcal{U} for the pattern calculus. We start by defining the sets of **types** and **multiset types**, given by means of the following grammars:

$$\begin{array}{ll}
\text{(Product Types)} \quad \mathcal{P} & ::= \times(\mathcal{A}_1, \mathcal{A}_2) \\
\text{(Types)} \quad \sigma & ::= \bullet \mid \mathcal{P} \mid \mathcal{A} \rightarrow \sigma \\
\text{(Multiset Types)} \quad \mathcal{A} & ::= [\sigma_k]_{k \in K}
\end{array}$$

where \bullet is an atomic type, K is a (possibly empty) finite set of indexes, and a multiset type is an unordered list of (not necessarily different) elements, where $[]$ denotes the *empty* multiset. We write $|\mathcal{A}|$ to denote the number of elements of the multiset \mathcal{A} . For example $[\bullet, [] \rightarrow \bullet, \bullet]$ is a multiset type of 3 elements, representing the intersection type $(\bullet \cap ([] \rightarrow \bullet)) \cap \bullet$, where \cap is an associative, commutative and non-idempotent intersection type constructor. We write \sqcup to denote multiset union. Multiset types are used to specify how programs consume terms: intuitively, the empty multiset is assigned to terms that are erased during (head) reduction, while duplicable terms are necessarily typed with non-empty multisets. As usual the arrow type is right-associative.

A product type, representing the type of a pair, is defined as the product of two (possibly empty) multisets of types. This formulation of product types turns out to be a key tool in our quantitative framework, and constitutes an essential difference with the product types proposed in [19], which are modeled by disjoint unions, so that any pair $\langle t, u \rangle$ of *typed* terms t and u has necessarily *at least* two types, one of the form $\times_1(\sigma)$ where σ is the type of t , and one of the form $\times_2(\tau)$, where τ is the type of u . Indeed, in op. cit., multiset types carry two completely different meanings: being a pair (but not necessarily a pair to be duplicated), or being a duplicable term (but not necessarily a pair). Our specification of products can then be interpreted as the use of the exponential isomorphism $!(A \wp B) \equiv !A \otimes !B$ of multiplicative exponential linear logic [28].

A **typing context** Γ is a map from variables to multiset types, such that only finitely many variables are not mapped to the empty multiset $[]$. We write $\text{dom}(\Gamma)$ to denote the domain of Γ , which is the set $\{x \mid \Gamma(x) \neq []\}$. We may write $\Gamma \# \Delta$ if and only if $\text{dom}(\Gamma)$ and $\text{dom}(\Delta)$ are disjoint. Given typing contexts $\{\Gamma_i\}_{i \in I}$ we write $\bigwedge_{i \in I} \Gamma_i$ for the context that maps x to $\sqcup_{i \in I} \Gamma_i(x)$. One particular case is $\Gamma \wedge \Delta$. We sometimes write $\Gamma; \Delta$ instead of $\Gamma \wedge \Delta$, when $\Gamma \# \Delta$, and we do not distinguish $\Gamma; x : []$ from Γ . The typing context $\Gamma|_p$ is such that $\Gamma|_p(x) = \Gamma(x)$, if $x \in \text{var}(p)$ and $[]$ otherwise. The typing context $\Gamma \parallel \mathcal{V}$ is defined by $(\Gamma \parallel \mathcal{V})(x) = \Gamma(x)$ if $x \notin \mathcal{V}$ and $[]$ otherwise. Finally, $\Gamma \subseteq \Delta$ means that $\text{dom}(\Gamma) \subseteq \text{dom}(\Delta)$ and $\Gamma(x) \sqsubseteq \Delta(x)$ for every $x \in \text{dom}(\Gamma)$, where \sqsubseteq denotes multiset inclusion.

The **type assignment system** \mathcal{U} is given in Fig. 2 and can be seen as a natural extension of Gardner's system [27] to explicit matching operators, pairs and product types. It assigns types (resp. multiset types) to terms, using an auxiliary (sub)system that assigns multiset types to patterns. We use $\Phi \triangleright \Gamma \vdash t : \sigma$ (resp. $\Phi \triangleright \Gamma \vdash t : \mathcal{A}$) to denote **term type derivations** ending with the sequent $\Gamma \vdash t : \sigma$ (resp. $\Gamma \vdash t : \mathcal{A}$), and $\Pi \triangleright \Gamma \Vdash p : \mathcal{A}$ to denote **pattern type derivations** ending with the sequent $\Gamma \Vdash p : \mathcal{A}$. The size of a derivation Φ , denoted by $\text{sz}(\Phi)$, is the number of all the typing rules used in Φ except **many**¹ (this is particularly appropriate in the proof of the substitution lemma).

Note that when assigning types (multiset types) to terms, we only allow the introduction of multiset types on the right through the **many** rule.

Most of the rules for terms are straightforward. Rule **match** is used to type the explicit matching operator $t[p \setminus u]$ and can be seen as a combination of rules **app** and **abs**. Rule **pat_v** is used when the pattern is a variable x . Its multiset type is the type declared for x in the typing context. Rule **pat_x** is used when the pattern has a product type, which means that the pattern will be matched with a pair. The condition $p \# q$ ensures linearity of patterns. Note that any pair term can be typed, in particular, with $\times([], [])$.

The system enjoys the key property of relevance:

¹ An equivalent type system can be presented without the **many** rule, for example [19]. However, the inductive proofs in the current presentation turn to be more elegant.

$\frac{}{x : \mathcal{A} \Vdash x : \mathcal{A}} \text{ (pat}_v\text{)}$	$\frac{\Gamma \Vdash p : \mathcal{A} \quad \Delta \Vdash q : \mathcal{B} \quad p \# q}{\Gamma \wedge \Delta \Vdash \langle p, q \rangle : [\times(\mathcal{A}, \mathcal{B})]} \text{ (pat}_\times\text{)}$
$\frac{}{x : [\sigma] \vdash x : \sigma} \text{ (ax)}$	$\frac{(\Gamma_k \vdash t : \sigma_k)_{k \in K}}{\wedge_{k \in K} \Gamma_k \vdash t : [\sigma_k]_{k \in K}} \text{ (many)}$
$\frac{\Gamma \vdash t : \sigma \quad \Gamma _p \Vdash p : \mathcal{A}}{\Gamma \Vdash \text{var}(p) \vdash \lambda p. t : \mathcal{A} \rightarrow \sigma} \text{ (abs)}$	$\frac{\Gamma \vdash t : \mathcal{A} \rightarrow \sigma \quad \Delta \vdash u : \mathcal{A}}{\Gamma \wedge \Delta \vdash t u : \sigma} \text{ (app)}$
$\frac{\Gamma \vdash t : \mathcal{A} \quad \Delta \vdash u : \mathcal{B}}{\Gamma \wedge \Delta \vdash \langle t, u \rangle : \times(\mathcal{A}, \mathcal{B})} \text{ (pair)}$	$\frac{\Gamma \vdash t : \sigma \quad \Gamma _p \Vdash p : \mathcal{A} \quad \Delta \vdash u : \mathcal{A}}{(\Gamma \Vdash \text{var}(p)) \wedge \Delta \vdash t[p \setminus u] : \sigma} \text{ (match)}$

■ **Figure 2** Typing System \mathcal{U} .

► **Lemma 4** (Relevance). *Let $\Phi \triangleright \Gamma \vdash t : \sigma$. Then, $\text{dom}(\Gamma) \subseteq \text{fv}(t)$.*

Proof. By induction on Φ (cf. App. A). ◀

Moreover, typing is stable by reduction and expansion, and the size of derivations is decreasing (resp. strictly decreasing) for \rightarrow_p reduction (resp. \rightarrow_h reduction).

► **Lemma 5.** *Let $\Phi \triangleright \Gamma \vdash t : \sigma$. Then,*

1. (Upper Subject Reduction). *$t \rightarrow_p t'$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ s.t. $\text{sz}(\Phi) \geq \text{sz}(\Phi')$, and $t \rightarrow_h t'$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ s.t. $\text{sz}(\Phi) > \text{sz}(\Phi')$.*
2. (Upper Subject Expansion). *$t' \rightarrow_p t$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ such that $\text{sz}(\Phi') \geq \text{sz}(\Phi)$ and $t' \rightarrow_h t$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ such that $\text{sz}(\Phi') > \text{sz}(\Phi)$.*

Proof. By induction on Φ , item 1 (resp. item 2) uses a substitution (resp. anti-substitution) lemma (see Lem.22 and Lem. 23 in App. A for details). ◀

Typed terms are (head) clash-free, i.e. they cannot head reduce to a clash.

► **Lemma 6** (Clash-Free). *Let $\Phi \triangleright \Gamma \vdash t : \sigma$. Then t is (head) clash-free.*

Proof. By induction on Φ (cf. App. A). ◀

Although the system in [19] already characterises head-normalisation in the pattern calculus, it does not provide upper bounds for the length of the head strategy. This is mainly due to the fact that the reduction system in [19] does not always decrease the measure of the typed terms, even when reduction is performed in the so-called *typed* occurrences. We can recover this situation, as witnessed by the following soundness and completeness result:

► **Theorem 7** (Characterisation of Head-Normalisation and Upper Bounds). *Let t be a term in the pattern calculus. Then (1) t is typable in system \mathcal{U} iff (2) t is head-normalisable iff (3) t is head-terminating. Moreover, if $\Phi \triangleright \Gamma \vdash t : \sigma$, then the head-strategy terminates on t in at most $\text{sz}(\Phi)$ steps.*

Proof. The statement (1) \Rightarrow (3) holds by upper subject reduction (Lem. 5.1) for \rightarrow_h . The statement (3) \Rightarrow (2) is straightforward since \rightarrow_h is included in \rightarrow_p . Finally, the statement (2) \Rightarrow (1) holds by the fact that canonical terms are typable (easy), and by using upper subject expansion for \rightarrow_p (Lem. 5.2). \blacktriangleleft

The previous upper bound result is especially possible thanks to the upper subject reduction property, stating in particular that reduction \rightarrow_h *strictly* decreases the size of typing derivations. It is worth noticing that the reduction relation in [19] does not enjoy this property, particularly in the case of the rule $t[p \setminus v]u \rightarrow (tu)[p \setminus v]$, which is a permuting conversion rule, (slightly) changing the structure of the type derivation, but not its size.

4 Towards a Relational Model for the Pattern Calculus

Denotational and operational semantics have tended to abstract quantitative information (e.g. time and space) as computational resource consumption. Since the invention of Girard's linear logic [28], where formulas are interpreted as resources, quantitative interpretation of programs, such as relational models [17, 18, 22], have been naturally defined and studied by following the simple idea that multisets are used to record the number of times a resource is consumed. Thus, relational models for the λ -calculus use multisets to keep track of how many times a resource is used during a computation.

In this brief section we emphasize a semantical result that is implicit in the previous section. Since relational models are often presented by means of typing systems [48, 47], our system \mathcal{U} suggests a quantitative model for our pair pattern calculus in the following way. Indeed, consider a term t such that $\text{fv}(t) \subseteq \{x_1, \dots, x_n\}$, in which case we say that the list $\vec{x} = (x_1, \dots, x_n)$ is **suitable** for t . Then, given $\vec{x} = (x_1, \dots, x_n)$ suitable for t , define the interpretation of a term t for \vec{x} as

$$\llbracket t \rrbracket_{\vec{x}} = \{((\mathcal{A}_1, \dots, \mathcal{A}_n), \sigma) \mid \text{there exists } \Phi \triangleright x_1 : \mathcal{A}_1, \dots, x_n : \mathcal{A}_n \vdash t : \sigma\}$$

A straightforward corollary of upper subject reduction and expansion properties (Lem. 5.1 and Lem. 5.2, respectively) is that $t =_p u$ implies $\llbracket t \rrbracket_{\vec{x}} = \llbracket u \rrbracket_{\vec{x}}$, where $=_p$ is the equational theory generated by the reduction relation \rightarrow_p . Thus, p -equivalent programs have the same meaning.

5 The \mathcal{E} Typing System

In this section we introduce our second typing system \mathcal{E} for the pattern calculus, which is obtained by refining the System \mathcal{U} presented in Sec. 3.

$$\begin{aligned} \text{(Product Types)} \quad \mathcal{P} &::= \times(\mathcal{A}_1, \mathcal{A}_2) \\ \text{(Tight Types)} \quad \mathfrak{t} &::= \bullet_{\mathcal{N}} \mid \bullet_{\mathcal{M}} \\ \text{(Types)} \quad \sigma &::= \mathfrak{t} \mid \mathcal{P} \mid \mathcal{A} \rightarrow \sigma \\ \text{(Multiset Types)} \quad \mathcal{A} &::= [\sigma_k]_{k \in K} \end{aligned}$$

Types in \mathfrak{t} , which can be seen as a refinement of the base type \bullet of System \mathcal{U} , denote the so-called tight types. The constant $\bullet_{\mathcal{M}}$ denotes the type of any term head reducing to a canonical form, while $\bullet_{\mathcal{N}}$ denotes the type of any term head reducing to a pure canonical form. We write $\text{tight}(\sigma)$, if σ is of the form $\bullet_{\mathcal{M}}$ or $\bullet_{\mathcal{N}}$ (we use \bullet to denote either form). We extend this notion to multisets of types and typing contexts as expected, that is, $\text{tight}([\sigma_i]_{i \in I})$ if $\text{tight}(\sigma_i)$ for all $i \in I$, and $\text{tight}(\Gamma)$ if $\text{tight}(\Gamma(x))$, for all $x \in \text{dom}(\Gamma)$.

The crucial idea behind the grammar of types is to distinguish between *consuming* constructors typed with standard types, and *persistent* constructors typed with tight types, as hinted in the introduction. A constructor is consuming (resp. persistent) if it is consumed (resp. not consumed) during head-reduction. Indeed, the pair constructor is consumed (on the pattern side as well as on the term side) during the execution of the pattern matching rule \mathbf{m} . Otherwise, patterns and pairs are persistent, and they do appear in the normal form of the original term. This dichotomy between consuming and persistent constructors, inspired from [41, 42], is reflected in the typing system by using different typing rules to type them, notably for the abstraction, the application, the pair terms and the pair patterns.

The **type assignment system** \mathcal{E} , given in Fig. 3, is based on sequents for terms (resp. patterns) with *counters* having the form $\Gamma \vdash^{(b,e,m,f)} t : \sigma$ or $\Gamma \vdash^{(b,e,m,f)} t : \mathcal{A}$ (resp. $\Gamma \Vdash^{(e,m,f)} p : \mathcal{A}$). Intuitively, if $\Gamma \vdash^{(b,e,m,f)} t : \sigma$ is “tightly” derivable (defined below), then $t \rightarrow_{\mathbf{h}}^{(b,e,m)} v$, where b is the number of **b**-steps, e the number of **e**-steps, m the number of **m**-steps and f is the size of the head normal-form v . Similarly, the derivability of $\Gamma \Vdash^{(e,m,f)} p : \mathcal{A}$ means that the pattern p generates e substitution **e**-steps, m matching **m**-steps and f symbols contributing to the normal form.

We write $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$ (resp. $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \mathcal{A}$) to denote **term type derivations** ending with the sequent $\Gamma \vdash^{(b,e,m,f)} t : \sigma$ (resp. $\Gamma \vdash^{(b,e,m,f)} t : \mathcal{A}$), and $\Pi \triangleright \Gamma \Vdash^{(e,m,f)} p : \mathcal{A}$ to denote **pattern type derivations** ending with the sequent $\Gamma \Vdash^{(e,m,f)} p : \mathcal{A}$. Often in examples, we will use the notation $\Phi^{(b,e,m,f)}$ (resp. $\Pi^{(e,m,f)}$) to refer to a term derivation (resp. pattern derivation) ending with a sequent annotated with indexes (b, e, m, f) (resp. (e, m, f)).

As mentioned in the introduction, exact bounds can only be extractable from *minimal* derivations. In our framework this notion is implemented by means of tightness [2]. We say that a derivation $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$ (resp. $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \mathcal{A}$) is **tight**, denoted by $\mathbf{tight}(\Phi)$, if and only if $\mathbf{tight}(\Gamma)$ and $\mathbf{tight}(\sigma)$ (resp. $\mathbf{tight}(\mathcal{A})$). The size of derivations is defined as in System \mathcal{U} .

We now give some intuition behind the typing rules in Fig. 3, by addressing in particular the consuming/persistent paradigm.

- Rule **ax**: Since x is itself a head normal-form, it will not generate any **b**, **e** or **m** steps, and its size is 0.
- Rule **abs**: Used to type abstractions $\lambda p.t$ to be applied (i.e. consumed), therefore it has a functional type $\mathcal{A} \rightarrow \sigma$. Final indexes of the abstraction are obtained from the ones of the body and the pattern, and 1 is added to the first index since the abstraction will be consumed by a **b**-reduction step.
- Rule **abs_p**: Used to type abstractions $\lambda p.t$ that are not going to be applied/consumed (they are persistent). Only the last index (size of the normal form) is incremented by one since the abstraction remains in the normal form (the abstraction is persistent). Note that both the body t and the variables in p should be typed with a tight type.
- Rule **app**: Types applications tu where t will eventually become an abstraction, and thus the application constructor will be consumed. Indexes for tu are exactly the sum of the indexes for t and u . Note that we do not need to increment the counter for **b** steps, since this was already taken into account in the **abs** rule.
- Rule **app_p**: Types applications tu where t is neutral, therefore will never become an abstraction, and the application constructor becomes persistent. Indexes are the ones for t , adding one to the (normal term) size to count for the (persistent) application.
- Rule **pair**: Types pairs consumed during some matching step. We add the indexes for the two components of the pair without incrementing the number of **m** steps, since it is incremented when typing a consuming abstraction, with rule **abs**.

$$\begin{array}{c}
\frac{}{x : \mathcal{A} \Vdash^{(1,0,0)} x : \mathcal{A}} \text{(pat}_v\text{)} \\
\frac{\Gamma \Vdash^{(e_p, m_p, n_p)} p : \mathcal{A} \quad \Delta \Vdash^{(e_q, m_q, n_q)} q : \mathcal{B} \quad p \# q}{\Gamma \wedge \Delta \Vdash^{(e_p+e_q, 1+m_p+m_q, n_p+n_q)} \langle p, q \rangle : [\times(\mathcal{A}, \mathcal{B})]} \text{(pat}_\times\text{)} \\
\frac{\text{dom}(\Gamma) \subseteq \text{var}(\langle p, q \rangle) \quad \text{tight}(\Gamma)}{\Gamma \Vdash^{(0,0,1)} \langle p, q \rangle : [\bullet\mathcal{N}]} \text{(pat}_p\text{)} \\
\hline
\frac{}{x : [\sigma] \vdash^{(0,0,0,0)} x : \sigma} \text{(ax)} \\
\frac{\Gamma \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma \quad \Gamma|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A}}{\Gamma \Vdash \text{var}(p) \vdash^{(b_t+1, e_t+e_p, m_t+m_p, f_t+f_p)} \lambda p. t : \mathcal{A} \rightarrow \sigma} \text{(abs)} \\
\frac{\Gamma \vdash^{(b, e, m, f)} t : \mathfrak{t} \quad \text{tight}(\Gamma|_p)}{\Gamma \Vdash \text{var}(p) \vdash^{(b, e, m, f+1)} \lambda p. t : \bullet\mathcal{M}} \text{(abs}_p\text{)} \\
\frac{(\Gamma_k \vdash^{(b_k, e_k, m_k, f_k)} t : \sigma_k)_{k \in K}}{\bigwedge_{k \in K} \Gamma_k \vdash^{(+k \in K b_k, +k \in K e_k, +k \in K m_k, +k \in K f_k)} t : [\sigma_k]_{k \in K}} \text{(many)} \\
\frac{\Gamma \vdash^{(b_t, e_t, m_t, f_t)} t : \mathcal{A} \rightarrow \sigma \quad \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}}{\Gamma \wedge \Delta \vdash^{(b_t+b_u, e_t+e_u, m_t+m_u, f_t+f_u)} t u : \sigma} \text{(app)} \\
\frac{\Gamma \vdash^{(b_t, e_t, m_t, f_t)} t : \bullet\mathcal{N}}{\Gamma \vdash^{(b_t, e_t, m_t, f_t+1)} t u : \bullet\mathcal{N}} \text{(app}_p\text{)} \\
\frac{\Gamma \vdash^{(b_t, e_t, m_t, f_t)} t : \mathcal{A} \quad \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{B}}{\Gamma \wedge \Delta \vdash^{(b_t+b_u, e_t+b_u, m_t+m_u, f_t+f_u)} \langle t, u \rangle : \times(\mathcal{A}, \mathcal{B})} \text{(pair)} \\
\frac{}{\vdash^{(0,0,0,1)} \langle t, u \rangle : \bullet\mathcal{M}} \text{(pair}_p\text{)} \\
\frac{\Gamma \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma \quad \Gamma|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A} \quad \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}}{(\Gamma \Vdash \text{var}(p)) \wedge \Delta \vdash^{(b_t+b_u, e_t+e_u+e_p, m_t+m_u+m_p, f_t+f_u+f_p)} t[p \setminus u] : \sigma} \text{(match)}
\end{array}$$

■ **Figure 3** Typing System \mathcal{E} .

- Rule pair_p : Used to type pairs that are not consumed in a matching step (they are persistent), therefore appear in the head normal-form. Since the pair is already a head normal-form its indexes are zero except for the size, which counts the pair itself.
- Rule match : Note that we do not need separate cases for consuming and persistent explicit matchings, since in both cases typable occurrences of u represent potential head reduction steps for u , which need to be taken into account in the final counter of the term.

3:14 A Quantitative Understanding of Pattern Matching

- Rule pat_v : Typed variables always generate one e and zero m steps, even when erased.
- Rule pat_\times : Used when the pattern has a product type, which means that the pattern will be matched with a pair. We add the counters for the two components of the pair and increment the counter for the m steps.
- Rule pat_p : Used when the pattern has a tight type, which means that it will not be matched with a pair and therefore will be blocked (it is persistent). This kind of pairs generate zero e and m steps, and will contribute with one blocked pattern to the size of the normal form.

The system also enjoys the relevance and clash-free properties, easily proved by induction:

- **Lemma 8** (Relevance). *Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$. Then, $\text{dom}(\Gamma) \subseteq \text{fv}(t)$.*
- **Lemma 9** (Clash-Free). *Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$. Then, t is (head) clash-free.*

We now discuss two examples.

- **Example 10.** Let us consider $t_0 = (\lambda\langle x, y \rangle. (\lambda\langle w, z \rangle. w y z) x) \langle \langle K, a \rangle, b \rangle$, with the following head-reduction sequence:

$$\begin{array}{ll}
 (\lambda\langle x, y \rangle. (\lambda\langle w, z \rangle. w y z) x) \langle \langle K, a \rangle, b \rangle & \rightarrow_b ((\lambda\langle w, z \rangle. w y z) x) [\langle x, y \rangle \setminus \langle \langle K, a \rangle, b \rangle] \\
 \rightarrow_b (w y z) [\langle w, z \rangle \setminus x] [\langle x, y \rangle \setminus \langle \langle K, a \rangle, b \rangle] & \rightarrow_m (w y z) [\langle w, z \rangle \setminus x] [x \setminus \langle K, a \rangle] [y \setminus b] \\
 \rightarrow_e (w y z) [\langle w, z \rangle \setminus \langle K, a \rangle] [y \setminus b] & \rightarrow_m (w y z) [w \setminus K] [z \setminus a] [y \setminus b] \\
 \rightarrow_e (K y z) [z \setminus a] [y \setminus b] & \rightarrow_b ((\lambda y_1. x_1) [x_1 \setminus y] z) [z \setminus a] [y \setminus b] \\
 \rightarrow_b x_1 [y_1 \setminus z] [x_1 \setminus y] [z \setminus a] [y \setminus b] & \rightarrow_e x_1 [x_1 \setminus y] [z \setminus a] [y \setminus b] \\
 \rightarrow_e y [z \setminus a] [y \setminus b] & \rightarrow_e y [y \setminus b] \\
 \rightarrow_e b &
 \end{array}$$

Note that, there are two matching steps in the head-reduction sequence, but the second step is only created after the substitution of x by $\langle K, a \rangle$. Our method allows us to extract this information from the typing derivations because of the corresponding types for $\langle x, y \rangle$ and $\langle w, z \rangle$. Indeed, both patterns are typed with a product type (cf. the forthcoming tight typing derivations), and therefore the corresponding pairs are consumed and not persistent.

Since $t_0 = (\lambda\langle x, y \rangle. (\lambda\langle w, z \rangle. w y z) x) \langle \langle K, a \rangle, b \rangle \rightarrow_h^{(4,6,2)} b$, the term t_0 should be tightly typable with counter $(4, 6, 2, 0)$, where 0 is the size of b . In the construction of such tight derivation we proceed by pieces. Let $T_K = [\bullet_N] \rightarrow [] \rightarrow \bullet_N$. We first construct the following pattern derivation for $\langle w, z \rangle$:

$$\Pi_{\langle w, z \rangle} \triangleright \frac{w : [T_K] \Vdash^{(1,0,0)} w : [T_K] \quad \Vdash^{(1,0,0)} z : []}{w : [T_K] \Vdash^{(2,1,0)} \langle w, z \rangle : [\times([T_K], [])]}$$

In the following $T_{\langle w, z \rangle} = [\times([T_K], [])]$. We construct a similar pattern derivation for $\langle x, y \rangle$:

$$\Pi_{\langle x, y \rangle} \triangleright \frac{x : T_{\langle w, z \rangle} \Vdash^{(1,0,0)} x : T_{\langle w, z \rangle} \quad y : [\bullet_N] \Vdash^{(1,0,0)} y : [\bullet_N]}{x : T_{\langle w, z \rangle}; y : [\bullet_N] \Vdash^{(2,1,0)} \langle x, y \rangle : [\times(T_{\langle w, z \rangle}, [\bullet_N])]}$$

In the rest of the example $T_{\langle x, y \rangle} = [\times(T_{\langle w, z \rangle}, [\bullet_N])]$. We build a type derivation for $\lambda\langle x, y \rangle. (\lambda\langle w, z \rangle. w y z) x$, where $\Gamma_w = w : [T_K]$, $\Gamma_y = y : [\bullet_N]$, $\Gamma = \Gamma_w; \Gamma_y$, and $\Gamma_x = x : T_{\langle w, z \rangle}$. Furthermore, in this example and throughout the paper, we will use $(\bar{0})$ to denote the tuple $(0, 0, 0, 0)$.

$$\begin{array}{c}
\frac{\Gamma_w \vdash^{(\bar{0})} w : T_K \quad \Gamma_y \vdash^{(\bar{0})} y : [\bullet_{\mathcal{N}}]}{\Gamma \vdash^{(\bar{0})} wy : [] \rightarrow \bullet_{\mathcal{N}} \quad \vdash^{(\bar{0})} z : []} \\
\frac{\Gamma \vdash^{(\bar{0})} wyz : \bullet_{\mathcal{N}} \quad \Pi_{\langle w, z \rangle}^{(2,1,0)}}{\Gamma_y \vdash^{(1,2,1,0)} \lambda \langle w, z \rangle . wyz : T_{\langle w, z \rangle} \rightarrow \bullet_{\mathcal{N}} \quad \Gamma_x \vdash^{(\bar{0})} x : T_{\langle w, z \rangle}} \\
\Phi_1 \triangleright \frac{\Gamma_y; \Gamma_x \vdash^{(1,2,1,0)} (\lambda \langle w, z \rangle . wyz)x : \bullet_{\mathcal{N}} \quad \Pi_{\langle x, y \rangle}^{(2,1,0)}}{\vdash^{(2,4,2,0)} \lambda \langle x, y \rangle . (\lambda \langle w, z \rangle . wyz)x : T_{\langle x, y \rangle} \rightarrow \bullet_{\mathcal{N}}} \\
\frac{x_1 : [\bullet_{\mathcal{N}}] \vdash^{(\bar{0})} x_1 : \bullet_{\mathcal{N}} \quad \Vdash^{(1,0,0)} y_1 : []}{x_1 : [\bullet_{\mathcal{N}}] \vdash^{(1,1,0,0)} \lambda y_1 . x_1 : [] \rightarrow \bullet_{\mathcal{N}} \quad x_1 : [\bullet_{\mathcal{N}}] \Vdash^{(1,0,0)} x_1 : [\bullet_{\mathcal{N}}]} \\
\Phi_K \triangleright \frac{\vdash^{(2,2,0,0)} K : T_K}{\vdash^{(2,2,0,0)} K : T_K}
\end{array}$$

From Φ_1 and Φ_K we build the final tight derivation for $t_0 = (\lambda \langle x, y \rangle . (\lambda \langle w, z \rangle . wyz)x) \langle \langle K, a \rangle, b \rangle$:

$$\begin{array}{c}
\frac{\Phi_K^{(2,2,0,0)}}{\vdash^{(2,2,0,0)} K : [T_K] \quad \vdash^{(\bar{0})} a : []} \\
\frac{\vdash^{(2,2,0,0)} \langle K, a \rangle : \times([T_K], [])}{\vdash^{(2,2,0,0)} \langle K, a \rangle : T_{\langle w, z \rangle} \quad b : [\bullet_{\mathcal{N}}] \vdash^{(\bar{0})} b : [\bullet_{\mathcal{N}}]} \\
\frac{b : [\bullet_{\mathcal{N}}] \vdash^{(2,2,0,0)} \langle \langle K, a \rangle, b \rangle : \times(T_{\langle w, z \rangle}, [\bullet_{\mathcal{N}}])}{\Phi_1^{(2,4,2,0)} \quad b : [\bullet_{\mathcal{N}}] \vdash^{(2,2,0,0)} \langle \langle K, a \rangle, b \rangle : [\times(T_{\langle w, z \rangle}, [\bullet_{\mathcal{N}}])]} \\
\Phi \triangleright \frac{\Phi_1^{(2,4,2,0)}}{b : [\bullet_{\mathcal{N}}] \vdash^{(4,6,2,0)} (\lambda \langle x, y \rangle . (\lambda \langle w, z \rangle . wyz)x) \langle \langle K, a \rangle, b \rangle : \bullet_{\mathcal{N}}}
\end{array}$$

Therefore, $\Phi^{(4,6,2,0)}$ gives the expected exact bounds. It is worth noticing that the pair $\langle \langle K, a \rangle, b \rangle$ is typed here with a singleton multiset, while it would be typable with a multiset having at least two elements in the typing system proposed in [19], even if the term is not going to be duplicated.

► **Example 11.** We now consider the term $t_1 = (\lambda z . (\lambda \langle x, y \rangle . \mathbf{I})zz) \langle u, v \rangle$, having the following head-reduction sequence to head normal-form:

$$\begin{array}{ll}
(\lambda z . (\lambda \langle x, y \rangle . \mathbf{I})zz) \langle u, v \rangle & \rightarrow_{\mathbf{b}} ((\lambda \langle x, y \rangle . \mathbf{I})zz)[z \setminus \langle u, v \rangle] \\
\rightarrow_{\mathbf{b}} (\mathbf{I}[\langle x, y \rangle \setminus z][z \setminus \langle u, v \rangle]) & \rightarrow_{\mathbf{b}} w[w \setminus z][\langle x, y \rangle \setminus z][z \setminus \langle u, v \rangle] \\
\rightarrow_{\mathbf{e}} z[\langle x, y \rangle \setminus z][z \setminus \langle u, v \rangle] & \rightarrow_{\mathbf{e}} \langle u, v \rangle[\langle x, y \rangle \setminus \langle u, v \rangle] \\
\rightarrow_{\mathbf{m}} \langle u, v \rangle[x \setminus u][y \setminus v] & \rightarrow_{\mathbf{e}} \langle u, v \rangle[y \setminus v] \\
\rightarrow_{\mathbf{e}} \langle u, v \rangle &
\end{array}$$

We have 3 **b**-steps, 4 **e**-steps, and 1 **m**-step to the normal form $\langle u, v \rangle$ of size 1. Note that the pair $\langle u, v \rangle$ is copied twice during the reduction, but only one of the copies is consumed by a matching. The copy of the pair that is not consumed will persist in the term, therefore it will be typed with $\bullet_{\mathcal{M}}$. The other copy will be consumed in a matching step, however its components are not going to be used, therefore we will type it with $[o]$, where o denotes $\times([], [])$.

Since $t_1 = (\lambda z . (\lambda \langle x, y \rangle . \mathbf{I})zz) \langle u, v \rangle \rightarrow_{\mathbf{h}}^{(3,4,1)} \langle u, v \rangle$, we need to derive a tight derivation for t_1 decorated with counter $(3, 4, 1, 1)$. We first consider the following derivation:

$$\begin{array}{c}
\frac{w : [\bullet_{\mathcal{M}}] \vdash^{(\bar{0})} w : \bullet_{\mathcal{M}} \quad w : [\bullet_{\mathcal{M}}] \Vdash^{(1,0,0)} w : [\bullet_{\mathcal{M}}] \quad \Vdash^{(1,0,0)} x : [] \quad \Vdash^{(1,0,0)} y : []}{\vdash^{(1,1,0,0)} \mathbf{I} : [\bullet_{\mathcal{M}}] \rightarrow \bullet_{\mathcal{M}} \quad \Vdash^{(2,0,0)} \langle x, y \rangle : [o]} \\
\Phi_1 \triangleright \frac{\vdash^{(1,1,0,0)} \mathbf{I} : [\bullet_{\mathcal{M}}] \rightarrow \bullet_{\mathcal{M}} \quad \Vdash^{(2,0,0)} \langle x, y \rangle : [o]}{\vdash^{(2,3,1,0)} \lambda \langle x, y \rangle . \mathbf{I} : [o] \rightarrow [\bullet_{\mathcal{M}}] \rightarrow \bullet_{\mathcal{M}}}
\end{array}$$

3:16 A Quantitative Understanding of Pattern Matching

From Φ_1 we obtain the following derivation Φ_2 , where $\mathcal{A}_0 = [o, \bullet\mathcal{M}]$:

$$\Phi_2 \triangleright \frac{\frac{\frac{\Phi_1^{(2,3,1,0)} \quad z : [o] \vdash^{(\bar{0})} z : [o]}{z : [o] \vdash^{(2,3,1,0)} (\lambda\langle x, y \rangle. \mathbf{I})z : [\bullet\mathcal{M}] \rightarrow \bullet\mathcal{M}} \quad z : [\bullet\mathcal{M}] \vdash^{(\bar{0})} z : [\bullet\mathcal{M}]}{z : \mathcal{A}_0 \vdash^{(2,3,1,0)} (\lambda\langle x, y \rangle. \mathbf{I})zz : \bullet\mathcal{M}} \quad z : \mathcal{A}_0 \vdash^{(1,0,0)} z : \mathcal{A}_0}{\vdash^{(3,4,1,0)} \lambda z. (\lambda\langle x, y \rangle. \mathbf{I})zz : \mathcal{A}_0 \rightarrow \bullet\mathcal{M}}}$$

Using Φ_2 we obtain the final tight derivation, and its expected counter:

$$\Phi_2^{(3,4,1,0)} \frac{\frac{\vdots}{\vdash^{(\bar{0})} \langle u, v \rangle : o} \quad \vdash^{(0,0,0,1)} \langle u, v \rangle : \bullet\mathcal{M}}{\vdash^{(0,0,0,1)} \langle u, v \rangle : \mathcal{A}_0} \quad \vdash^{(3,4,1,1)} (\lambda z. (\lambda\langle x, y \rangle. \mathbf{I})zz) \langle u, v \rangle : \bullet\mathcal{M}}$$

6 Soundness of System \mathcal{E}

This section studies the implication “tight typability implies head-normalisable”. The two key properties used to show this implication are *minimal counters for canonical forms* (Lem. 13) and the *exact subject reduction property* (Lem. 15). Indeed, Lem. 13 guarantees that a tight derivation for a canonical form t holds the right counter of the form $(0, 0, 0, |t|)$. Lem. 15 gives in fact an (*exact*) *weighted subject reduction* property, weighted because head-reduction strictly decreases the counters of typed terms, and exact because only *one* counter is decreased by 1 for each head-reduction step. Subject reduction is based on a *substitution* property (Lem. 14). We start with a key auxiliary lemma.

► **Lemma 12** (Tight Spreading). *Let $t \in \mathcal{N}$. Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$ be a typing derivation such that $\text{tight}(\Gamma)$. Then σ is tight and the last rule of Φ does not belong to $\{\text{app}, \text{abs}, \text{abs}_p, \text{pair}, \text{pair}_p\}$.*

Proof. By induction on $t \in \mathcal{N}$, taking into account the fact that t is not an abstraction nor a pair (cf. App. B). ◀

► **Lemma 13** (Canonical Forms and Minimal Counters). *Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$ be a tight derivation. Then $t \in \mathcal{M}$ if and only if $b = e = m = 0$.*

Proof. The left-to-right implication is by induction on the definition of the set \mathcal{M} , using the tight spreading property (Lem. 12) for the cases of application and explicit matching. The right-to-left implication is by induction on Φ and also uses Lem. 12 (cf. App. B). ◀

► **Lemma 14** (Substitution for System \mathcal{E}). *If $\Phi_t \triangleright \Gamma; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma$, and $\Phi_u \triangleright \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, then there exists $\Phi_{t\{x \setminus u\}} \triangleright \Gamma \wedge \Delta \vdash^{(b_t + b_u, e_t + e_u, m_t + m_u, f_t + f_u)} t\{x \setminus u\} : \sigma$.*

Proof. By induction on Φ_t (cf. App. B). ◀

► **Lemma 15** (Exact Subject Reduction). *If $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$, and $t \rightarrow_h t'$ is an s -step, with $s \in \{b, e, m\}$, then $\Phi' \triangleright \Gamma \vdash^{(b',e',m',f)} t' : \sigma$, where*

- $s = b$ implies $b' = b - 1$, $e' = e$, $m' = m$.
- $s = e$ implies $b' = b$, $e' = e - 1$, $m' = m$.
- $s = m$ implies $b' = b$, $e' = e$, $m' = m - 1$.

Proof. By induction on \rightarrow_h , using the substitution property (Lem. 14) (cf. App. B). ◀

The exact subject reduction property provides a simple argument to obtain the implication “tightly typable implies head-normalisable”: if t is tightly typable, and reduction decreases the counters, then head-reduction necessarily terminates. But the soundness implication is in fact more precise than that. Indeed:

► **Theorem 16** (Soundness). *Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$ be a tight derivation. Then there exists $u \in \mathcal{M}$ and a head reduction sequence ρ such that $\rho : t \rightarrow_{\mathfrak{h}}^{(b,e,m)} u$ and $|u| = f$.*

Proof. By induction on $b + e + m$.

If $b + e + m = 0$ (i.e. $b = e = m = 0$), then canonical forms and minimal counters property (Lem. 13) gives $t \in \mathcal{M}$, so that $t \not\rightarrow_{\mathfrak{h}}$ holds by Prop. 3. We let $u := t$ and thus $t \rightarrow_{\mathfrak{h}}^{(0,0,0)} t$. It is easy to show that tight derivations $\Phi \triangleright \Gamma \vdash^{(0,0,0,f)} t : \sigma$ for terms in \mathcal{M} verify $|t| = f$.

If $b + e + m > 0$, we know by Lem. 13 that $t \notin \mathcal{M}$, and we know by the clash-free property (Lem. 9) that t is (head) clash-free. Then, t turns to be head-reducible by Prop. 3, i.e. there exists t' such that $t \rightarrow_{\mathfrak{h}} t'$. By the exact subject reduction property (Lem. 15) there is a derivation $\Phi' \triangleright \Gamma \vdash^{(b',e',m',f)} t' : \sigma$ such that $b' + e' + m' + 1 = b + e + m$. The i.h. applied to Φ' then gives $t' \rightarrow_{\mathfrak{h}}^{(b',e',m')} u$ and $|u| = f$. We conclude with the sequence $t \rightarrow_{\mathfrak{h}} t' \rightarrow_{\mathfrak{h}}^{(b',e',m')} u$, with the counters as expected. ◀

7 Completeness for System \mathcal{E}

In this section we study the reverse implication “head-normalisable implies tight typability”. In this case the key properties are the existence of *tight derivations for canonical forms* (Lem. 17) and the *subject expansion property* (Lem. 19). As in the previous section these properties are (*exact*) *weighted* in the sense that Lem. 17 guarantees that a canonical form t has a tight derivation with the right counter, and Lem. 19 shows that each step of head-expansion strictly increases exactly one of the counters of tightly typed terms. Subject expansion relies on an *anti-substitution* property (Lem. 18).

► **Lemma 17** (Canonical Forms and Tight Derivations). *Let $t \in \mathcal{M}$. There exists a tight derivation $\Phi \triangleright \Gamma \vdash^{(0,0,0,|t|)} t : \mathfrak{t}$.*

Proof. We generalise the property to the two following statements:

- If $t \in \mathcal{N}$, then there exists a tight derivation $\Phi \triangleright \Gamma \vdash^{(0,0,0,|t|)} t : \bullet_{\mathcal{N}}$.
- If $t \in \mathcal{M}$, then there exists a tight derivation $\Phi \triangleright \Gamma \vdash^{(0,0,0,|t|)} t : \mathfrak{t}$.

The proof then proceeds by induction on \mathcal{N}, \mathcal{M} , using relevance (Lem. 8). ◀

► **Lemma 18** (Anti-Substitution for System \mathcal{E}). *Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t\{x \setminus u\} : \sigma$. Then, there exist derivations Φ_t, Φ_u , integers $b_t, b_u, e_t, e_u, m_t, m_u, f_t, f_u$, contexts Γ_t, Γ_u , and multitype \mathcal{A} such that $\Phi_t \triangleright \Gamma_t; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, $b = b_t + b_u$, $e = e_t + e_u$, $m = m_t + m_u$, $f = f_t + f_u$, and $\Gamma = \Gamma_t \wedge \Gamma_u$.*

Proof. By induction on Φ (cf. App. C). ◀

► **Lemma 19** (Exact Subject Expansion). *If $\Phi' \triangleright \Gamma \vdash^{(b',e',m',f')} t' : \sigma$, and $t \rightarrow_{\mathfrak{h}} t'$ is an s -step, with $s \in \{\mathfrak{b}, \mathfrak{e}, \mathfrak{m}\}$, then $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$, where*

- $s = \mathfrak{b}$ implies $b = b' + 1$, $e' = e$, $m' = m$.
- $s = \mathfrak{e}$ implies $b' = b$, $e = e' + 1$, $m' = m$.
- $s = \mathfrak{m}$ implies $b' = b$, $e' = e$, $m = m' + 1$.

Proof. By induction on $\rightarrow_{\mathfrak{h}}$, using the anti-substitution property (Lem. 18) (cf. App. C). ◀

The previous lemma provides a simple argument to obtain the implication “head-normalisable implies tightly typable”, which can in fact be stated in a more precise way:

► **Theorem 20 (Completeness).** *Let t be a head-normalising term such that $t \rightarrow_h^{(b,e,m)} u$, $u \in \mathcal{M}$. Then there exists a tight derivation $\Phi \triangleright \Gamma \vdash^{(b,e,m,|u|)} t : \tau$.*

Proof. By induction on $b + e + m$.

- If $(b + e + m) = 0$ then $t = u \in \mathcal{M}$, therefore $\Gamma \vdash^{(0,0,0,|t|)} t : \tau$, by the canonical forms and tight derivations property (Lem. 17).
- If $(b + e + m) > 0$, then $t \rightarrow_h t' \rightarrow_h^{(b',e',m')} u$, where $b' + e' + m' + 1 = b + e + m$. By the i.h. $\Gamma \vdash^{(b',e',m',|u|)} t' : \tau$. Then from the exact subject expansion property (Lem. 19), it follows that $\Gamma \vdash^{(b,e,m,|u|)} t : \tau$. ◀

In summary, soundness and completeness do not only establish an equivalence between tight typability and head-normalisation, but they provide a much refined equivalence property stated as follows:

► **Corollary 21.** *Given a term t , the following statements are equivalent*

- *There is a tight derivation $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \tau$.*
- *There exists a canonical form $u \in \mathcal{M}$ such that $t \rightarrow_h^{(b,e,m)} u$ and $|u| = f$.*

8 Conclusion

This paper provides a quantitative insight of pattern matching by using type systems to study some of its *dynamical* properties. Indeed, our typing system \mathcal{U} (resp. \mathcal{E}) provides upper bounds (resp. exact measures) about time and space properties related to (dynamic) computation. More precisely, the tuple of integers in the conclusion of a *tight* \mathcal{E} -derivation for a term t provides the exact *length* of the head-normalisation sequence of t and the *size* of its normal form. Moreover, the length of the normalisation sequence is *discriminated* according to different kind of steps performed to evaluate t .

Future work includes generalisations to more powerful notions of (dynamic) patterns, and to other reduction strategies for pattern calculi, as well to programs with recursive schemes. Inhabitation for our typing system is conjectured to be decidable, as the one in [19], but this still needs to be formally proved, in which case the result “solvability = typing+ inhabitation” in opt. cit. would be restated in a simpler framework. The quest of a general notion of model for pattern calculi also remains open, particularly for dynamic pattern calculi [32, 6].

Last, but not least, time cost analysis of a language with constructors and pattern matching is studied in [1], where it is shown that evaluation matching rules other than β -reduction may be negligible, depending on the reduction strategy and the specific notion of value. We expect the type-based quantitative technical tools we provide in this paper to be helpful in such a kind of quantitative analysis.

References

- 1 Beniamino Accattoli and Bruno Barras. The negligible and yet subtle cost of pattern matching. In *APLAS*, volume 10695 of *LNCS*, pages 426–447. Springer, 2017.
- 2 Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *PACMPL*, 2(ICFP):94:1–94:30, 2018.
- 3 Beniamino Accattoli and Giulio Guerrieri. Types of fireballs. In *APLAS*, volume 11275 of *LNCS*, pages 45–66. Springer, 2018.

- 4 Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. Types by need. In *ESOP*, volume 11423 of *LNCSS*, pages 410–439. Springer, 2019.
- 5 Beniamino Accattoli and Delia Kesner. Preservation of strong normalisation modulo permutations for the structural lambda-calculus. *Logical Methods in Computer Science*, 8(1), 2012. doi:10.2168/LMCS-8(1:28)2012.
- 6 Sandra Alves, Besik Dundua, Mário Florido, and Temur Kutsia. Pattern-based calculi with finitary matching. *Logic Journal of the IGPL*, 26(2):203–243, 2018.
- 7 Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *PACMPL*, 1(ICFP):43:1–43:29, 2017.
- 8 Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. Foundations of strong call by need. *PACMPL*, 1(ICFP):20:1–20:29, 2017.
- 9 Pablo Barenbaum, Eduardo Bonelli, and Kareem Mohamed. Pattern matching and fixed points: Resource types and strong call-by-need: Extended abstract. In *PPDP*, pages 6:1–6:12. ACM Press, 2018.
- 10 Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundation of mathematics*. North-Holland, Amsterdam, revised edition, 1984.
- 11 Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.
- 12 Alexis Bernadet. *Types intersections non-idempotents pour raffiner la normalisation forte avec des informations quantitatives*. PhD thesis, École Polytechnique, 2014.
- 13 Alexis Bernadet and Stéphane Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4), 2013.
- 14 Frédéric Blanqui. Size-based termination of higher-order rewriting. *Journal of Functional Programming*, 28:e11, 2018.
- 15 Eduardo Bonelli, Delia Kesner, Carlos Lombardi, and Alejandro Ríos. Normalisation for dynamic pattern calculi. In *RTA*, volume 15 of *LIPICs*, pages 117–132. Schloss Dagstuhl, 2012.
- 16 Gérard Boudol, Pierre-Louis Curien, and Carolina Lavatelli. A semantics for lambda calculi with resources. *Mathematical Structures in Computer Science*, 9(4):437–482, 1999.
- 17 Antonio Bucciarelli and Thomas Ehrhard. On phase semantics and denotational semantics: the exponentials. *Annals of Pure and Applied Logic*, 109(3):205–241, 2001.
- 18 Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. A relational semantics for parallelism and non-determinism in a functional setting. *Annals of Pure and Applied Logic*, 163(7):918–934, 2012.
- 19 Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Observability for pair pattern calculi. In *TLCA*, volume 38 of *LIPICs*, pages 123–137, 2015.
- 20 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.
- 21 Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–463, 2001.
- 22 Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. PhD thesis, Université Aix-Marseille II, 2007.
- 23 Daniel de Carvalho. Execution time of λ -terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.
- 24 Daniel de Carvalho and Lorenzo Tortora de Falco. A semantic account of strong normalization in linear logic. *Information and Computation*, 248:104–129, 2016.
- 25 Thomas Ehrhard. Collapsing non-idempotent intersection types. In *CSL*, volume 16 of *LIPICs*, pages 259–273. Schloss Dagstuhl, 2012.
- 26 Thomas Ehrhard and Giulio Guerrieri. The bang calculus: an untyped lambda-calculus generalizing call-by-name and call-by-value. In *PPDP*, pages 174–187. ACM Press, 2016.
- 27 Philippa Gardner. Discovering needed reductions using type theory. In *TACS*, volume 789 of *LNCSS*, pages 555–574. Springer, 1994.

- 28 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.
- 29 Giulio Guerrieri and Giulio Manzonetto. The bang calculus and the two girard’s translations. In *Linearity-TLLA*, volume 292 of *EPTCS*, pages 15–30, 2018.
- 30 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In *CAV*, volume 7358 of *LNCS*, pages 781–786. Springer, 2012.
- 31 Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *POPL*, pages 359–373. ACM Press, 2017.
- 32 Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.
- 33 Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL*, pages 223–236. ACM Press, 2010.
- 34 Steffen Jost, Pedro B. Vasconcelos, Mário Florido, and Kevin Hammond. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning*, 59(1):87–120, 2017.
- 35 Wolfram Kahl. Basic pattern matching calculi: a fresh view on matching failure. In *FLOPS*, volume 2998 of *LNCS*, pages 276–290. Springer, 2004.
- 36 Delia Kesner. Reasoning about call-by-need by means of types. In *FoSSaCS*, volume 9634 of *LNCS*, pages 424–441. Springer, 2016.
- 37 Delia Kesner, Carlos Lombardi, and Alejandro Ríos. Standardisation for constructor based pattern calculi. In *HOR*, volume 49, page 58–72, 2011.
- 38 Delia Kesner and Daniel Ventura. Quantitative types for the linear substitution calculus. In *IFIP TCS*, volume 8705 of *LNCS*, pages 296–310. Springer, 2014.
- 39 Delia Kesner and Daniel Ventura. A resource aware computational interpretation for Herbelin’s syntax. In *ICTAC*, volume 9399 of *LNCS*, pages 388–403. Springer, 2015.
- 40 Delia Kesner and Pierre Vial. Types as resources for classical natural deduction. In *FSCD*, volume 84 of *LIPICs*, pages 24:1–24:17. Schloss Dagstuhl, 2017.
- 41 Delia Kesner and Pierre Vial. Extracting exact bounds from typing in a classical framework. 25th International Conference on Types for Proofs and Programs, 2019.
- 42 Delia Kesner and Pierre Vial. Consuming and persistent types for classical logic. In *LICS*. IEEE Computer Society, 2020.
- 43 Assaf Kfoury. A linearization of the lambda-calculus and consequences. *Journal of Logic and Computation*, 10(3):411–436, 2000.
- 44 Jean Louis Krivine. *Lambda-Calculus, Types and Models*. Masson, Paris, and Ellis Horwood, Hemel Hempstead, 1993.
- 45 Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, 8(4), 2011.
- 46 Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. *Science of Computer Programming*, 84:77–100, 2014.
- 47 C.-H. Luke Ong. Quantitative semantics of the lambda calculus: Some generalisations of the relational model. In *LICS*, pages 1–12. IEEE Computer Society, 2017.
- 48 Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Essential and relational models. *Mathematical Structures in Computer Science*, 27(5):626–650, 2017.
- 49 Álvaro J. Rebón Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro B. Vasconcelos. Cost analysis using automatic size and time inference. In *IFL*, volume 2670 of *LNCS*, pages 232–248. Springer, 2002.
- 50 RaML. Resource Aware ML. URL: <http://raml.co>.
- 51 Franz Sigmüller. Type-based resource analysis on haskell. In *DICE-FOPARA*, volume 298 of *EPTCS*, pages 47–60, 2019.
- 52 Hugo R. Simões, Kevin Hammond, Mário Florido, and Pedro B. Vasconcelos. Using intersection types for cost-analysis of higher-order polymorphic functional programs. In *TYPES*, volume 4502 of *LNCS*, pages 221–236. Springer, 2006.

- 53 Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *ICFP*, pages 165–176. ACM Press, 2012.
- 54 Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *IFL*, volume 3145 of *LNCS*, pages 86–101. Springer, 2003.
- 55 Philip Wadler. The essence of functional programming. In *POPL*, pages 1–14. ACM Press, 1992.
- 56 Joe B. Wells. The essence of principal typings. In *ICALP*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

A The \mathcal{U} Typing System

► **Lemma 4** (Relevance). *Let $\Phi \triangleright \Gamma \vdash t : \sigma$. Then, $\text{dom}(\Gamma) \subseteq \text{fv}(t)$.*

Proof. Let $\Phi \triangleright \Gamma \vdash t : \sigma$. By straightforward induction on Φ . Note that $\Gamma = (\Gamma \parallel \text{var}(p)); \Gamma|_p$ in both (abs) and (match) rules. ◀

► **Lemma 22** (Substitution for System \mathcal{U}). *If $\Phi_t \triangleright \Gamma; x : \mathcal{A} \vdash t : \sigma$, and $\Phi_u \triangleright \Delta \vdash u : \mathcal{A}$, then there exists $\Phi_{t\{x\backslash u\}} \triangleright \Gamma \wedge \Delta \vdash t\{x\backslash u\} : \sigma$ such that $\text{sz}(\Phi_{t\{x\backslash u\}}) = \text{sz}(\Phi_t) + \text{sz}(\Phi_u) - |\mathcal{A}|$.*

Proof. We generalise the statement as follows: Let $\Phi_u \triangleright \Delta \vdash u : \mathcal{A}$.

- If $\Phi_t \triangleright \Gamma; x : \mathcal{A} \vdash t : \sigma$, then there exists $\Phi_{t\{x\backslash u\}} \triangleright \Gamma \wedge \Delta \vdash t\{x\backslash u\} : \sigma$.
- If $\Phi_t \triangleright \Gamma; x : \mathcal{A} \vdash t : \mathcal{B}$, then there exists $\Phi_{t\{x\backslash u\}} \triangleright \Gamma \wedge \Delta \vdash t\{x\backslash u\} : \mathcal{B}$.

In both cases $\text{sz}(\Phi_{t\{x\backslash u\}}) = \text{sz}(\Phi_t) + \text{sz}(\Phi_u) - |\mathcal{A}|$.

The proof then follows by induction on Φ_t .

- If Φ_t is (ax), then we consider two cases:
 - $t = x$: then $\Phi_x \triangleright x : [\sigma] \vdash x : \sigma$ and $\Phi_u \triangleright \Delta \vdash u : [\sigma]$, which is a consequence of $\Delta \vdash u : \sigma$. Then $x\{x\backslash u\} = u$, and we trivially obtain $\Phi_{t\{x\backslash u\}} \triangleright \Delta \vdash u : \sigma$. We have $\text{sz}(\Phi_{t\{x\backslash u\}}) = 1 + \text{sz}(\Phi_u) - 1$ as expected.
 - $t = y$: then $\Phi_y \triangleright y : [\sigma]; x : [] \vdash y : \sigma$ and $\Phi_u \triangleright \emptyset \vdash u : []$ by the (many) rule. Then $y\{x\backslash u\} = y$, and we trivially obtain $\Phi_{t\{x\backslash u\}} \triangleright y : [\sigma] \vdash y : \sigma$. We have $\text{sz}(\Phi_{t\{x\backslash u\}}) = 1 + 0 - 0$ as expected.
- If Φ_t ends with (many), then it has premises of the form $(\Phi_t^i \triangleright \Gamma^i; x : \mathcal{A}^i \vdash t : \sigma^i)_{i \in I}$, where $\Gamma = \bigwedge_{i \in I} \Gamma^i$, $\mathcal{A} = \bigwedge_{i \in I} \mathcal{A}^i$ and $\mathcal{B} = [\sigma^i]_{i \in I}$. The derivation Φ_u can also be decomposed into subderivations $(\Phi_u^i \triangleright \Delta^i \vdash u : \mathcal{A}^i)_{i \in I}$ where $\Delta = \bigwedge_{i \in I} \Delta^i$. The i.h. gives the derivations $(\Phi_{t\{x\backslash u\}}^i \triangleright \Gamma^i \wedge \Delta^i \vdash t\{x\backslash u\} : \sigma^i)_{i \in I}$. Then we apply rule (many) to get $\Phi_{t\{x\backslash u\}} \triangleright \Gamma \wedge \Delta \vdash t\{x\backslash u\} : \mathcal{B}$. The statement about $\text{sz}(_)$ works as expected by the i.h.
- If Φ_t ends with (abs), so that $t = \lambda p.t'$ then, without loss of generality, one can always assume that $(\text{fv}(u) \cup \{x\}) \cap \text{var}(p) = \emptyset$. The result will follow easily by induction and relevance of the typing system. The statement about $\text{sz}(_)$ works as expected by the i.h.
- If Φ_t ends with (app), so that $t = t'u'$, then $\Phi_{t'u'}$ is of the form

$$\frac{\Phi_{t'} \triangleright \Gamma_{t'}; x : \mathcal{A}_{t'} \vdash t' : \mathcal{B} \rightarrow \sigma \quad \Phi_{u'} \triangleright \Gamma_{u'}; x : \mathcal{A}_{u'} \vdash u' : \mathcal{B}}{\Gamma_{t'} \wedge \Gamma_{u'}; x : \mathcal{A}_{t'} \wedge \mathcal{A}_{u'} \vdash t'u' : \sigma}$$

Also, $\Phi_u \triangleright \Delta \vdash u : \mathcal{A}$ is a consequence of $(\Delta_k \vdash u : \sigma_k)_{k \in K}$, with $\mathcal{A} = [\sigma_k]_{k \in K}$ and $\Delta = \bigwedge_{k \in K} \Delta_k$. Note that $\mathcal{A} = \mathcal{A}_{t'} \wedge \mathcal{A}_{u'} = [\sigma_i]_{i \in K_{t'}} \wedge [\sigma_i]_{i \in K_{u'}}$, with $K = K_{t'} \uplus K_{u'}$, from which one can obtain both $\Delta_{t'} \vdash u : \mathcal{A}_{t'}$ and $\Delta_{u'} \vdash u : \mathcal{A}_{u'}$, through the (many) rule. By

the i.h. we then have $\Gamma_{t'} \wedge \Delta_{t'} \vdash t'\{x \setminus u\} : \mathcal{B} \rightarrow \sigma$ and $\Gamma_{u'} \wedge \Delta_{u'} \vdash u'\{x \setminus u\} : \mathcal{B}$. Finally, $\Gamma_{t'} \wedge \Gamma_{u'} \wedge \Delta_{t'} \wedge \Delta_{u'} \vdash (t'\{x \setminus u\})(u'\{x \setminus u\}) : \sigma$ by the **app** rule. The statement about $\mathbf{sz}(_)$ works as expected by the i.h.

- If Φ_t ends with **(pair)** or **(pair_p)**, so that $t = \langle t', u' \rangle$, then the result is obtained by induction following the same reasoning used in rule **app**. The statement about $\mathbf{sz}(_)$ works as expected by the i.h.
- If Φ_t ends with **(match)**, so that $t = t'[p \setminus u']$, then the proof is similar to the application case since $t'[p \setminus u']\{x \setminus u\} = (t'\{x \setminus u\})[p \setminus u'\{x \setminus u\}]$ and we can assume that $(\mathbf{fv}(u) \cup \{x\}) \cap \mathbf{var}(p) = \emptyset$. The statement about $\mathbf{sz}(_)$ works as expected by the i.h. ◀

► **Lemma 23** (Anti-Substitution for System \mathcal{U}). *Let $\Phi \triangleright \Gamma \vdash t\{x \setminus u\} : \sigma$. Then, there exist derivations Φ_t, Φ_u , contexts Γ_t, Γ_u , and multitype \mathcal{A} such that $\Phi_t \triangleright \Gamma_t; x : \mathcal{A} \vdash t : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$ and $\Gamma = \Gamma_t \wedge \Gamma_u$. Moreover, $\mathbf{sz}(\Phi) = \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Phi_u) - |\mathcal{A}|$.*

Proof. As in the case of the substitution lemma, the proof follows by generalising the property for the two cases where the type derivation Φ assigns a type or a multiset type:

- Let $\Phi \triangleright \Gamma \vdash t\{x \setminus u\} : \sigma$. Then, there exist derivations Φ_t, Φ_u , contexts Γ_t, Γ_u , and multitype \mathcal{A} such that $\Phi_t \triangleright \Gamma_t; x : \mathcal{A} \vdash t : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$ and $\Gamma = \Gamma_t \wedge \Gamma_u$.
- Let $\Phi \triangleright \Gamma \vdash t\{x \setminus u\} : \mathcal{B}$. Then, there exist derivations Φ_t, Φ_u , contexts Γ_t, Γ_u , and multitype \mathcal{A} such that $\Phi_t \triangleright \Gamma_t; x : \mathcal{A} \vdash t : \mathcal{B}$, $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$ and $\Gamma = \Gamma_t \wedge \Gamma_u$.

In both cases $\mathbf{sz}(\Phi) = \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Phi_u) - |\mathcal{A}|$ holds.

We will reason by induction on Φ and cases analysis on t . For all the rules (except **many**), we will have the trivial case $t\{x \setminus u\}$, where $t = x$, in which case $t\{x \setminus u\} = u$, for which we have a derivation $\Phi \triangleright \Gamma \vdash u : \sigma$. Therefore $\Phi_t \triangleright x : [\sigma] \vdash x : \sigma$ and $\Phi_u \triangleright \Gamma \vdash u : [\sigma]$ is obtained from Φ using the **(many)** rule. We conclude since $\mathbf{sz}(\Phi) = 1 + \mathbf{sz}(\Phi_u) - 1$. We now reason on the different cases assuming that $t \neq x$.

- If Φ is **(ax)** then $\Phi \triangleright y : [\sigma] \vdash y : \sigma$ and, since $t \neq x$, $t = y \neq x$. Then we take $\mathcal{A} = []$, $\Phi_t \triangleright y : [\sigma]; x : [] \vdash y : \sigma$, and $\Phi_u \triangleright \emptyset \vdash u : []$ from rule **(many)**. We conclude since $\mathbf{sz}(\Phi) = 1 + 0 - 0$.
- If Φ ends with **(many)**, then $\Phi \triangleright \bigwedge_{k \in K} \Gamma_k \vdash t\{x \setminus u\} : [\sigma_k]_{k \in K}$ follows from the derivation $\Phi^k \triangleright \Gamma_k \vdash t\{x \setminus u\} : \sigma_k$, for each $k \in K$. By the i.h. there exist Φ_t^k, Φ_u^k , contexts Γ_t^k, Γ_u^k and multitype \mathcal{A}_k , such that $\Phi_t^k \triangleright \Gamma_t^k; x : \mathcal{A}_k \vdash t : \sigma_k$, $\Phi_u^k \triangleright \Gamma_u^k \vdash u : \mathcal{A}_k$, $\Gamma_k = \Gamma_t^k \wedge \Gamma_u^k$. Taking $\mathcal{A} = \bigwedge_{k \in K} \mathcal{A}_k$ and using rule **many** we get $\bigwedge_{k \in K} \Gamma_t^k; x : \mathcal{A} \vdash t : [\sigma_k]_{k \in K}$. From the premises of Φ_u^k for $k \in K$, applying the **many** rule, we get $\bigwedge_{k \in K} \Gamma_u^k \vdash u : \mathcal{A}$. Note that $\Gamma = \bigwedge_{k \in K} \Gamma_k = (\bigwedge_{k \in K} \Gamma_t^k) \wedge (\bigwedge_{k \in K} \Gamma_u^k)$. The statement about $\mathbf{sz}(_)$ works as expected by the i.h.
- If Φ ends with **(abs)**, then $t = \lambda p.t'$, therefore $\Phi \triangleright \Gamma \setminus \mathbf{var}(p) \vdash \lambda p.(t'\{x \setminus u\}) : \mathcal{B} \rightarrow \sigma$ follows from $\Phi' \triangleright \Gamma \vdash t'\{x \setminus u\} : \sigma$ and $\Pi_p \triangleright \Gamma|_p \Vdash p : \mathcal{B}$. Note that, one can always assume that $\mathbf{var}(p) \cap \mathbf{fv}(u) = \emptyset$ and $x \notin \mathbf{var}(p)$. By the i.h., $\Phi_{t'} \triangleright \Gamma_{t'}; x : \mathcal{A} \vdash t' : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$, with $\Gamma = \Gamma_{t'} \wedge \Gamma_u$. Then using **abs** we get $\Phi_t \triangleright \Gamma_{t'} \setminus \mathbf{var}(p); x : \mathcal{A} \vdash \lambda p.t' : \mathcal{B} \rightarrow \sigma$. Note that $(\Gamma_{t'}; x : \mathcal{A}) \setminus \mathbf{var}(p) = \Gamma_{t'} \setminus \mathbf{var}(p); x : \mathcal{A}$ and $\Gamma \setminus \mathbf{var}(p) = (\Gamma_{t'} \setminus \mathbf{var}(p)) \wedge \Gamma_u$. The statement about $\mathbf{sz}(_)$ works as expected by the i.h.
- The remaining cases for **(app)**, **(pair)** and **(match)** also hold by the i.h. and do not present any special difficulty. ◀

► **Lemma 5.** Let $\Phi \triangleright \Gamma \vdash t : \sigma$. Then,

1. (Upper Subject Reduction). $t \rightarrow_p t'$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ s.t. $\mathbf{sz}(\Phi) \geq \mathbf{sz}(\Phi')$, and $t \rightarrow_h t'$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ s.t. $\mathbf{sz}(\Phi) > \mathbf{sz}(\Phi')$.
2. (Upper Subject Expansion). $t' \rightarrow_p t$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ such that $\mathbf{sz}(\Phi') \geq \mathbf{sz}(\Phi)$ and $t' \rightarrow_h t$ implies there is $\Phi' \triangleright \Gamma \vdash t' : \sigma$ such that $\mathbf{sz}(\Phi') > \mathbf{sz}(\Phi)$.

Proof. Let $\Phi \triangleright \Gamma \vdash t : \sigma$.

1. By induction on \rightarrow_p (resp. \rightarrow_h) and the substitution property (Lem. 22). The first

three cases represent the base cases for both reductions, where the size relation is strict.

- $t = \mathbb{L}[\lambda p.v]u \rightarrow_{p/h} \mathbb{L}[v[p \setminus u]] = t'$. The proof is by induction on the list \mathbb{L} . We only show the case of the empty list as the other one is straightforward. The typing derivation Φ is necessarily of the form

$$\frac{\frac{\Gamma_v \vdash v : \sigma \quad \Gamma_v|_p \Vdash p : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(p) \vdash \lambda p.v : \mathcal{A} \rightarrow \sigma} \quad \Gamma_u \vdash u : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \vdash (\lambda p.v)u : \sigma}$$

We then construct the following derivation Φ' :

$$\frac{\Gamma_v \vdash v : \sigma \quad \Gamma_v|_p \Vdash p : \mathcal{A} \quad \Gamma_u \vdash u : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \vdash v[p \setminus u] : \sigma}$$

Moreover, $\mathbf{sz}(\Phi) = \mathbf{sz}(\Phi') + 1$.

- $t = v[x \setminus u] \rightarrow_p v\{x \setminus u\} = t'$. Then Φ has two term premises $\Phi_v \triangleright \Gamma_v; x : \mathcal{A} \vdash v : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$, and one pattern premise $\Pi_x \triangleright x : \mathcal{A} \Vdash x : \mathcal{A}$, where $\Gamma = \Gamma_v \wedge \Gamma_u$ and $\mathbf{sz}(\Phi) = \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Phi_u) + \mathbf{sz}(\Pi_x) + 1$. Lem. 22 then gives a derivation Φ' ending with $\Gamma_v \wedge \Gamma_u \vdash v\{x \setminus u\} : \sigma$, where $|\mathcal{A}| \geq 0$ and $\mathbf{sz}(\Pi_x) = 1$ imply

$$\mathbf{sz}(\Phi') = \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Phi_u) - |\mathcal{A}| < \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Phi_u) + \mathbf{sz}(\Pi_x) < \mathbf{sz}(\Phi)$$

When $t = v[x \setminus u] \rightarrow_h v\{x \setminus u\} = t'$, where $v \not\rightarrow_h$, the same results hold.

- $t = v[\langle p_1, p_2 \rangle \setminus \mathbb{L}[\langle u_1, u_2 \rangle]] \rightarrow_p \mathbb{L}[v[p_1 \setminus u_1][p_2 \setminus u_2]] = t'$. Let us write $p = \langle p_1, p_2 \rangle$ and $u = \langle u_1, u_2 \rangle$. The typing derivation Φ is necessarily of the form

$$\frac{\Phi_v \triangleright \Gamma_v \vdash v : \sigma \quad \Pi_p \triangleright \Gamma_v|_p \Vdash \langle p_1, p_2 \rangle : \mathcal{A} \quad \Phi_u \triangleright \Gamma_u \vdash \mathbb{L}[\langle u_1, u_2 \rangle] : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(\langle p_1, p_2 \rangle) \wedge \Gamma_u \vdash v[\langle p_1, p_2 \rangle \setminus \mathbb{L}[\langle u_1, u_2 \rangle]] : \sigma}$$

Moreover, $\mathcal{A} = [\times(\mathcal{A}_1, \mathcal{A}_2)]$ and $\mathbf{sz}(\Phi) = \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_p) + \mathbf{sz}(\Phi_u) + 1$.

Then Π_p is of the form:

$$\frac{\Pi_{p_1} \triangleright \Gamma_v|_{p_1} \Vdash p_1 : \mathcal{A}_1 \quad \Pi_{p_2} \triangleright \Gamma_v|_{p_2} \Vdash p_2 : \mathcal{A}_2 \quad p_1 \# p_2}{\Gamma_v|_p \Vdash \langle p_1, p_2 \rangle : [\times(\mathcal{A}_1, \mathcal{A}_2)]}$$

and $\mathbf{sz}(\Pi_p) = \mathbf{sz}(\Pi_{p_1}) + \mathbf{sz}(\Pi_{p_2}) + 1$

The proof is then by induction on the list \mathbb{L} .

- For $\mathbb{L} = \square$ we have Φ_u of the form:

$$\frac{\frac{\Phi_{u_1} \triangleright \Gamma_{u_1} \vdash u_1 : \mathcal{A}_1 \quad \Phi_{u_2} \triangleright \Gamma_{u_2} \vdash u_2 : \mathcal{A}_2}{\Gamma_u \vdash \langle u_1, u_2 \rangle : \times(\mathcal{A}_1, \mathcal{A}_2)}}{\Gamma_u \vdash \langle u_1, u_2 \rangle : \mathcal{A}}$$

where $\Gamma_u = \Gamma_{u_1} \wedge \Gamma_{u_2}$ and $\mathbf{sz}(\Phi_u) = \mathbf{sz}(\Phi_{u_1}) + \mathbf{sz}(\Phi_{u_2}) + 1$. We first construct the following derivation:

$$\frac{\Phi_v \triangleright \Gamma_v \vdash v : \sigma \quad \Pi_{p_1} \triangleright \Gamma_v|_{p_1} \Vdash p_1 : \mathcal{A}_1 \quad \Phi_{u_1} \triangleright \Gamma_{u_1} \vdash u_1 : \mathcal{A}_1}{\Gamma_v \parallel \mathbf{var}(p_1) \wedge \Gamma_{u_1} \vdash v[p_1 \setminus u_1] : \sigma}$$

By using Lem. 4 and α -conversion, we construct a derivation Φ' with conclusion $\Gamma_v \parallel \mathbf{var}(p_1) \parallel \mathbf{var}(p_2) \wedge \Gamma_{u_1} \wedge \Gamma_{u_2} \vdash v[p_1 \setminus u_1][p_2 \setminus u_2] : \sigma$. Note that $p_1 \# p_2$ implies $\Gamma_v \parallel \mathbf{var}(\langle p_1, p_2 \rangle) = \Gamma_v \parallel \mathbf{var}(p_1) \parallel \mathbf{var}(p_2)$. Thus, we finally obtain $\mathbf{sz}(\Phi') = \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_p) + \mathbf{sz}(\Phi_u) < \mathbf{sz}(\Phi)$.

- Let $L = L'[q \setminus s]$. Then Φ_u is necessarily of the following form:

$$\frac{\frac{\Phi_{L'} \triangleright \Delta_u \vdash L'[\langle u_1, u_2 \rangle] : \times(\mathcal{A}_1, \mathcal{A}_2) \quad \Pi_q \triangleright \Delta_u|_q \Vdash q : \mathcal{B} \quad \Phi_s \triangleright \Delta_s \vdash s : \mathcal{B}}{\Gamma_u \vdash L'[\langle u_1, u_2 \rangle][q \setminus s] : \times(\mathcal{A}_1, \mathcal{A}_2)}}{\Gamma_u \vdash L'[\langle u_1, u_2 \rangle][q \setminus s] : \mathcal{A}}$$

where $\Gamma_u = \Delta_u \parallel \text{var}(q) \wedge \Delta_s$.

We will apply the i.h. on the reduction step $v[p \setminus L'[u]] \rightarrow_p L'[v[p_1 \setminus u_1][p_2 \setminus u_2]]$, in particular we type the left-hand side term with the following derivation Ψ_1 :

$$\frac{\frac{\Phi_{L'} \triangleright \Delta_u \vdash L'[\langle u_1, u_2 \rangle] : \times(\mathcal{A}_1, \mathcal{A}_2)}{\Delta_u \vdash L'[\langle u_1, u_2 \rangle] : \mathcal{A}}}{\Gamma_v \parallel \text{var}(p) \wedge \Delta_u \vdash v[\langle p_1, p_2 \rangle \setminus L'[\langle u_1, u_2 \rangle]] : \sigma}$$

The i.h. gives a derivation $\Psi_2 \triangleright \Gamma_v \parallel \text{var}(p) \wedge \Delta_u \vdash L'[v[p_1 \setminus u_1][p_2 \setminus u_2]] : \sigma$ verifying $\text{sz}(\Psi_2) < \text{sz}(\Psi_1)$. Let $\Lambda = \Gamma_v \parallel \text{var}(p) \wedge \Delta_u$. We conclude with the following derivation Φ' :

$$\frac{\Psi_2 \quad \Pi_q \triangleright \Delta_u|_q \Vdash q : \mathcal{B} \quad \Phi_s \triangleright \Delta_s \vdash s : \mathcal{B}}{\Lambda \parallel \text{var}(q) \wedge \Delta_s \vdash L'[v[p_1 \setminus u_1][p_2 \setminus u_2]][q \setminus s] : \sigma}$$

Indeed, we first remark that $\Lambda|_q = \Delta_u|_q$ holds by relevance and α -conversion. Secondly, $\Gamma_v \parallel \text{var}(p) \wedge \Gamma_u = \Gamma_v \parallel \text{var}(p) \wedge (\Delta_u \parallel \text{var}(q)) \wedge \Delta_s = \Lambda \parallel \text{var}(q) \wedge \Delta_s$ also holds by Lem. 4 and α -conversion. Last, we have

$$\begin{aligned} \text{sz}(\Phi') &= \text{sz}(\Psi_2) + \text{sz}(\Pi_q) + \text{sz}(\Phi_s) + 1 &< \\ &= \text{sz}(\Psi_1) + \text{sz}(\Pi_q) + \text{sz}(\Phi_s) + 1 &= \\ &= \text{sz}(\Phi_v) + \text{sz}(\Pi_p) + \text{sz}(\Phi_{L'}) + 1 + \text{sz}(\Pi_q) + \text{sz}(\Phi_s) + 1 &= \\ &= \text{sz}(\Phi_v) + \text{sz}(\Pi_p) + \text{sz}(\Phi_u) + 1 &= \text{sz}(\Phi) \end{aligned}$$

When $t = v[\langle p_1, p_2 \rangle \setminus L[\langle u_1, u_2 \rangle]] \rightarrow_h L[v[p_1 \setminus u_1][p_2 \setminus u_2]] = t'$, where $v \not\rightarrow_h$, the same results hold.

- Most of the inductive cases are straightforward. We only detail here two interesting cases.
 - $t = v[p \setminus u] \rightarrow_p v[p \setminus u'] = t'$, where $u \rightarrow_p u'$. The proof holds here by the i.h. In particular, when $p = x$ and $x \notin \text{fv}(v)$, then by relevance we have x of type $[]$ as well as u of type $[]$. This means that both u and u' are typed by a (many) rule with no premise, and in that case we get $\text{sz}(\Phi) = \text{sz}(\Phi')$.
 - $t = v[p \setminus u] \rightarrow_h v[p \setminus u'] = t'$, where $v \not\rightarrow_h$ and $p \neq x$ and $u \rightarrow_h u'$. By construction there are typing subderivations $\Phi_v \triangleright \Gamma_v \vdash v : \sigma$, $\Pi_p \triangleright \Gamma_v|_p \Vdash p : \mathcal{A}$ and $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$ such that $\Gamma = \Gamma_v \parallel \text{var}(p) \wedge \Gamma_u$. Since p is not a variable then Π_p ends with rule pat_\times . In which case \mathcal{A} contains exactly one type, let us say $\mathcal{A} = [\sigma_u]$. Then Φ_u has the following form

$$\frac{\Gamma_u \vdash u : \sigma_u}{\Phi_u \triangleright \Gamma_u \vdash u : [\sigma_u]}$$

The i.h. applied to the premise of Φ_u gives a derivation $\Gamma_u \vdash u' : \sigma_u$ and having the expected size relation. To conclude we build a type derivation Φ' for $v[p \setminus u']$ having the expected size relation.

2. By induction on \rightarrow_p (resp. \rightarrow_h) and the anti-substitution property (Lem. 23).
- $t' = L[\lambda p.v]u \rightarrow_{p/h} L[v[p\backslash u]] = t$. The proof is by induction on the list L . We consider the case $L = \square$, since the other case follows straightforward by i.h. The typing derivation Φ is necessarily of the form:

$$\frac{\Gamma_v \vdash v : \sigma \quad \Gamma_v|_p \Vdash p : \mathcal{A} \quad \Gamma_u \vdash u : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \vdash v[p\backslash u] : \sigma}$$

We then construct the following derivation Φ' :

$$\frac{\frac{\Gamma_v \vdash v : \sigma \quad \Gamma_v|_p \Vdash p : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(p) \vdash \lambda p.v : \mathcal{A} \rightarrow \sigma} \quad \Gamma_u \vdash u : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \vdash (\lambda p.v)u : \sigma}$$

Moreover, $\mathbf{sz}(\Phi') = \mathbf{sz}(\Phi) + 1$.

- $t' = v[x\backslash u] \rightarrow_p v\{x\backslash u\} = t$. Then by Lem. 23, there exist derivations Φ_v, Φ_u , contexts Γ_v, Γ_u and a multitype \mathcal{A} , such that $\Phi_v \triangleright \Gamma_v; x : \mathcal{A} \vdash v : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$, $\Gamma = \Gamma_v \wedge \Gamma_u$, and $\mathbf{sz}(\Phi) = \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Phi_u) - |\mathcal{A}|$. Furthermore, one has $\Pi_x \triangleright x : \mathcal{A} \Vdash x : \mathcal{A}$. Then one can construct the following derivation Φ' :

$$\frac{\Gamma_v; x : \mathcal{A} \vdash v : \sigma \quad x : \mathcal{A} \Vdash x : \mathcal{A} \quad \Gamma_u \vdash u : \mathcal{A}}{\Gamma_v \wedge \Gamma_u \vdash v[x\backslash u] : \sigma}$$

Furthermore, $\mathbf{sz}(\Phi') = \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_x) + \mathbf{sz}(\Phi_u) > \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Phi_u) - |\mathcal{A}|$, since $|\mathcal{A}| \geq 0$ and $\mathbf{sz}(\Pi_x) = 1$. The same result holds for $t = v[x\backslash u] \rightarrow_h v\{x\backslash u\} = t'$, where $v \not\rightarrow_h$.

- $t' = v[\langle p_1, p_2 \rangle \backslash L[\langle u_1, u_2 \rangle]] \rightarrow_p L[v[p_1\backslash u_1][p_2\backslash u_2]] = t$. Let us write $p = \langle p_1, p_2 \rangle$ and $u = \langle u_1, u_2 \rangle$. The proof is by induction on the list L .
 - $L = \square$, then the typing derivation Φ is necessarily of the form:

$$\frac{\frac{\Gamma_v \vdash v : \sigma \quad \Gamma_v|_{p_1} \Vdash p_1 : \mathcal{A}_1 \quad \Gamma_1 \vdash u_1 : \mathcal{A}_1}{(\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1 \vdash v[p_1\backslash u_1] : \sigma} \quad ((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1)|_{p_2} \Vdash p_2 : \mathcal{A}_2 \quad \Gamma_2 \vdash u_2 : \mathcal{A}_2}{((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1) \parallel \mathbf{var}(p_2) \wedge \Gamma_2 \vdash v[p_1\backslash u_1][p_2\backslash u_2] : \sigma}$$

where $\Gamma = (((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1) \parallel \mathbf{var}(p_2)) \wedge \Gamma_2$. Moreover, the following equality holds $((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1) \parallel \mathbf{var}(p_2) = ((\Gamma_v \parallel \mathbf{var}(p_1)) \parallel \mathbf{var}(p_2)) \wedge \Gamma_1 \parallel \mathbf{var}(p_2)$, since $(\Gamma_v \parallel \mathbf{var}(p_1)) \parallel \mathbf{var}(p_2) = \Gamma_v \parallel \mathbf{var}(p)$ and $\Gamma_1 \parallel \mathbf{var}(p_2) =_{L.4} \Gamma_1$. Similarly, $((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1)|_{p_2} =_{L.4} (\Gamma_v \parallel \mathbf{var}(p_1))|_{p_2}$ and, by linearity of patterns, we have $(\Gamma_v \parallel \mathbf{var}(p_1))|_{p_2} = \Gamma_v|_{p_2}$. Hence, we conclude with the following derivation Φ' :

$$\frac{\Gamma_v \vdash v : \sigma \quad \frac{\Gamma_v|_{p_1} \Vdash p_1 : \mathcal{A}_1 \quad \Gamma_v|_{p_2} \Vdash p_2 : \mathcal{A}_2}{\Gamma_v|_p \Vdash p : [\times(\mathcal{A}_1, \mathcal{A}_2)]} \quad \frac{\Gamma_1 \vdash u_1 : \mathcal{A}_1 \quad \Gamma_2 \vdash u_2 : \mathcal{A}_2}{\Gamma_1 \wedge \Gamma_2 \vdash u : [\times(\mathcal{A}_1, \mathcal{A}_2)]}}{\Gamma_v \wedge \Gamma_2 \vdash u : [\times(\mathcal{A}_1, \mathcal{A}_2)]} \quad \frac{\Gamma_v \parallel \mathbf{var}(p) \wedge (\Gamma_1 \wedge \Gamma_2) \vdash v[p\backslash u] : \sigma}{\Gamma_v \parallel \mathbf{var}(p) \wedge (\Gamma_1 \wedge \Gamma_2) \vdash v[p\backslash u] : \sigma}$$

Furthermore,

$$\begin{aligned} \mathbf{sz}(\Phi) &= \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_{p_1}) + \mathbf{sz}(\Phi_{u_1}) + 1 + \mathbf{sz}(\Pi_{p_2}) + \mathbf{sz}(\Phi_{u_2}) + 1 = \\ &= \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_{p_1}) + \mathbf{sz}(\Pi_{p_2}) + 1 + \mathbf{sz}(\Phi_{u_1}) + \mathbf{sz}(\Phi_{u_2}) + 1 < \\ &= \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_p) + \mathbf{sz}(\Phi_u) + 1 = \mathbf{sz}(\Phi') \end{aligned}$$

- If $L = L'[q \setminus s]$, then $t' = v[\langle p_1, p_2 \rangle \setminus L'[\langle u_1, u_2 \rangle]] \rightarrow_p L'[q \setminus s][v[p_1 \setminus u_1][p_2 \setminus u_2]] = L'[v[p_1 \setminus u_1][p_2 \setminus u_2]][q \setminus s] = t$, and Φ is of the form:

$$\frac{\Phi_{L'} \triangleright \Gamma_{L'} \vdash L'[v[p_1 \setminus u_1][p_2 \setminus u_2]] : \sigma \quad \Pi_q \triangleright \Gamma_{L'}|_q \Vdash q : \mathcal{A} \quad \Phi_s \triangleright \Gamma_s \vdash s : \mathcal{A}}{\Gamma_{L'} \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash L'[v[p_1 \setminus u_1][p_2 \setminus u_2]][q \setminus s] : \sigma}$$

From $v[\langle p_1, p_2 \rangle \setminus L'[\langle u_1, u_2 \rangle]] \rightarrow_p L'[v[p_1 \setminus u_1][p_2 \setminus u_2]]$ and $\Phi_{L'}$ by the i.h. one gets $\Phi'_{L'} \triangleright \Gamma_{L'} \vdash v[\langle p_1, p_2 \rangle \setminus L'[\langle u_1, u_2 \rangle]] : \sigma$ with $\mathbf{sz}(\Phi'_{L'}) > \mathbf{sz}(\Phi_{L'})$. Furthermore $\Phi'_{L'}$ is necessarily of the form:

$$\frac{\Phi_u \triangleright \Gamma_u \vdash L'[u] : \times(\mathcal{A}_1, \mathcal{A}_2) \quad \Phi_v \triangleright \Gamma_v \vdash v : \sigma \quad \Pi_p \triangleright \Gamma_v|_p \Vdash p : [\times(\mathcal{A}_1, \mathcal{A}_2)] \quad \Gamma_u \vdash L'[u] : [\times(\mathcal{A}_1, \mathcal{A}_2)]}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \vdash v[p \setminus L'[u]] : \sigma}$$

Then one can construct the following derivation Φ'_u :

$$\frac{\frac{\Gamma_u \vdash L'[u] : \times(\mathcal{A}_1, \mathcal{A}_2) \quad \Gamma_u|_q \Vdash q : \mathcal{A} \quad \Phi_s}{\Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash L'[q \setminus s][u] : \times(\mathcal{A}_1, \mathcal{A}_2)}}{\Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash L'[q \setminus s][u] : [\times(\mathcal{A}_1, \mathcal{A}_2)]}$$

From which we build Φ' :

$$\frac{\Gamma_v \vdash v : \sigma \quad \Gamma_v|_p \Vdash p : [\times(\mathcal{A}_1, \mathcal{A}_2)] \quad \Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash L'[q \setminus s][u] : [\times(\mathcal{A}_1, \mathcal{A}_2)]}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash v[p \setminus L'[q \setminus s][u]] : \sigma}$$

With $\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s = (\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u) \parallel \mathbf{var}(q) \wedge \Gamma_s = \Gamma$. Furthermore

$$\begin{aligned} \mathbf{sz}(\Phi) &= \mathbf{sz}(\Phi_{L'}) + \mathbf{sz}(\Pi_q) + \mathbf{sz}(\Phi_s) + 1 && \leq_{i.h.} \\ &= \mathbf{sz}(\Phi'_{L'}) + \mathbf{sz}(\Pi_q) + \mathbf{sz}(\Phi_s) + 1 && = \\ &= \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_p) + \mathbf{sz}(\Phi_u) + \mathbf{sz}(\Pi_q) + \mathbf{sz}(\Phi_s) + 1 + 1 && = \\ &= \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Pi_p) + \mathbf{sz}(\Phi'_u) + 1 && = \mathbf{sz}(\Phi') \end{aligned}$$

The same result holds for $t' = v[\langle p_1, p_2 \rangle \setminus L[\langle u_1, u_2 \rangle]] \rightarrow_p L[v[p_1 \setminus u_1][p_2 \setminus u_2]] = t$, where $v \not\rightarrow_h$.

- Most of the inductive cases are straightforward. We only detail here two interesting cases.
 - $t' = v[p \setminus u'] \rightarrow_p v[p \setminus u] = t$, where $u' \rightarrow_p u$. The proof holds by the i.h. In particular, when $p = x$ and $x \notin \mathbf{fv}(v)$, then by relevance we have x of type $[\]$ as well as u of type $[\]$. This means that u, u' are typed by a (many) rule with no premise, and in that case we get $\mathbf{sz}(\Phi) = \mathbf{sz}(\Phi')$.
 - $t' = v[p \setminus u'] \rightarrow_h v[p \setminus u] = t$, where $v \not\rightarrow_h$ and $p \neq x$ and $u' \rightarrow_h u$. By construction there are subderivations $\Phi_v \triangleright \Gamma_v \vdash v : \sigma$, $\Pi_p \triangleright \Gamma_v|_p \Vdash p : \mathcal{A}$ and $\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A}$ for some multiset \mathcal{A} and $\Gamma = (\Gamma_v \parallel \mathbf{var}(p)) \wedge \Gamma_u$. Since p is not a variable then Π_p ends with rule (\mathbf{pat}_\times) , in which case \mathcal{A} contains only one type, let us say $\mathcal{A} = [\sigma_u]$. Then Φ_u has the following form:

$$\Phi_u \triangleright \frac{\Gamma_u \vdash u : \sigma_u}{\Gamma_u \vdash u' : [\sigma_u]}$$

The i.h. applied to the premise of Φ_u gives a derivation $\Gamma_u \vdash u' : \sigma_u$. Therefore, we construct the following derivation Φ' :

$$\frac{\Gamma_v \vdash v : \sigma \quad \Gamma_v|_p \Vdash p : [\sigma_u] \quad \Gamma_u \vdash u' : \sigma_u}{\Gamma_v \parallel \text{var}(p) \wedge \Gamma_u \vdash v[p \setminus u'] : \sigma}$$

Furthermore,

$$\begin{aligned} \text{sz}(\Phi) &= \text{sz}(\Phi_v) + \text{sz}(\Pi_p) + \text{sz}(\Phi_u) + 1 <_{i.h.} \\ &\text{sz}(\Phi_v) + \text{sz}(\Pi_p) + \text{sz}(\Phi_{u'}) + 1 = \text{sz}(\Phi') \end{aligned} \quad \blacktriangleleft$$

► **Lemma 6** (Clash-Free). *Let $\Phi \triangleright \Gamma \vdash t : \sigma$. Then t is (head) clash-free.*

Proof. Let $\Phi \triangleright \Gamma \vdash t : \sigma$. By induction on $\text{sz}(\Phi)$, using the syntax-directed aspect of system \mathcal{U} .

- The base case for rule (ax) is trivial.
- The cases for rules (many) and (abs) are straightforward from the i.h.
- The case for (pair) is also straightforward since every pair is (head) clash-free.
- Let us consider the case for (match), where Φ has the following form:

$$\frac{\Gamma_t \vdash t : \sigma \quad \Gamma_t|_p \Vdash p : \mathcal{A} \quad \Delta \vdash u : \mathcal{A}}{(\Gamma_t \parallel \text{var}(p)) \wedge \Delta \vdash t[p \setminus u] : \sigma} \text{ (match)}$$

- If $t \rightarrow_h t'$ for some t' , so that $t[p \setminus u] \rightarrow_h t'[p \setminus u]$, then the size of the typing derivation of $t'[p \setminus u]$ is smaller than that of Φ by Upper Subject Reduction. The i.h. then gives $t'[p \setminus u]$ (head) clash-free and thus $t[p \setminus u]$ is (head) clash-free.
- If $t \not\rightarrow_h$ then there are two cases.
 - * If p is a variable x , then $t[x \setminus u] \rightarrow_h t\{x \setminus u\}$ and by Upper Subject Reduction $t\{x \setminus u\}$ has a type derivation strictly smaller than that of Φ , thus by the i.h. $t\{x \setminus u\}$ is (head) clash-free and so is $t[x \setminus u]$.
 - * Otherwise p is a pair, so that $\mathcal{A} \neq []$ (i.e. $\mathcal{A} = [\times(\mathcal{A}_1, \mathcal{A}_2)]$).
 - If $u \rightarrow_h u'$ then $t[p \setminus u] \rightarrow_h t[p \setminus u']$, so that the size of the typing derivation of $t[p \setminus u']$ is smaller than that of Φ by Upper Subject Reduction. The i.h. then gives $t[p \setminus u']$ (head) clash-free and thus $t[p \setminus u]$ is (head) clash-free.
 - If $u \not\rightarrow_h$ then $t[p \setminus u]$ is a head-normal form. In order to guarantee that $t[p \setminus u]$ is (head) clash-free note that u cannot be of the form $L[\lambda q.v]$, which can only be typed with a multiset of functional types.
- Let us consider the case for rule (app), where Φ has the following form

$$\frac{\Phi_u \triangleright \Gamma_u \vdash u : \mathcal{A} \rightarrow \sigma \quad \Phi_v \triangleright \Gamma_v \vdash v : \mathcal{A}}{\Gamma_u \wedge \Gamma_v \vdash uv : \sigma}$$

Note that u cannot be of the form $L[\langle u_1, u_2 \rangle]$ because it is typed with a functional type, thus it is either $L[x]$ or $L[\lambda p.u']$.

If u is $L[x]$, then u is (head) clash-free by the i.h. and thus uv is necessarily (head) clash-free.

If u is $L[\lambda p.u']$ then $t = L[\lambda p.u']v \rightarrow_h L[u'[p \setminus v]] = t'$ and the size of the type derivation of t' is strictly smaller than the size of Φ by Upper Subject Reduction. The i.h. gives t' (head) clash-free, and thus t is also (head) clash-free. ◀

B Soundness of System \mathcal{E}

► **Lemma 12** (Tight Spreading). *Let $t \in \mathcal{N}$. Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$ be a typing derivation such that $\text{tight}(\Gamma)$. Then σ is tight and the last rule of Φ does not belong to $\{\text{app}, \text{abs}, \text{abs}_p, \text{pair}, \text{pair}_p\}$.*

Proof. First note that, since $t \in \mathcal{N}$, then t is not an abstraction nor a pair, therefore one cannot apply any of the rules $\{\text{abs}, \text{abs}_p, \text{pair}, \text{pair}_p, \text{emptypair}\}$. We now examine the remaining rules.

- $t = x$. Then Φ is an axiom $x : [\mathfrak{t}] \vdash^{(0,0,0,0)} x : \mathfrak{t}$ so the property trivially holds.
- $t = uv$, with $u \in \mathcal{N}$. Then Φ has a (left) subderivation $\Phi_u \triangleright \Gamma_u \vdash^{(b,e,m,f)} u : \sigma_u$, and since $\Gamma_u \subseteq \Gamma$, then Γ_u is necessarily tight. Therefore, by the i.h., $\sigma_u = \bullet_{\mathcal{N}}$, from which follows that $\sigma = \bullet_{\mathcal{N}}$ by applying rule (app_p) . Note that one cannot apply rule (app) to type uv , since t would have to be an arrow type, which contradicts the i.h.
- $t = u[p \setminus v]$, with $u \in \mathcal{N}, v \in \mathcal{N}$. Then Φ follows from $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \sigma$, $\Phi_p \triangleright \Gamma_u|_p \vdash^{(e_p, m_p, f_p)} p : \mathcal{A}$ and $\Phi_v \triangleright \Delta \vdash^{(b_v, e_v, m_v, f_v)} v : \mathcal{A}$, where $\Gamma = (\Gamma_u \parallel \text{var}(p)) \wedge \Delta$. Since $(\Gamma_u \parallel \text{var}(p)) \wedge \Delta$ is tight, then Δ is tight. By the i.h. on v one gets a derivation $\Delta \vdash^{(b_v, e_v, m_v, f_v)} v : \bullet_{\mathcal{N}}$ so that $\Delta \vdash^{(b_v, e_v, m_v, f_v)} v : [\bullet_{\mathcal{N}}]$ follows from many and $\Gamma_u|_p \vdash^{(0,0,1)} p : [\bullet_{\mathcal{N}}]$ necessarily follows from rule (pat_p) . This implies $\Gamma_u|_p$ is tight, therefore Γ_u is tight. Since $u \in \mathcal{N}$ the i.h. gives $\sigma \in \mathfrak{t}$ as expected. ◀

► **Lemma 13** (Canonical Forms and Minimal Counters). *Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$ be a tight derivation. Then $t \in \mathcal{M}$ if and only if $b = e = m = 0$.*

Proof. By induction on $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$, where Φ is tight (right-to-left implication), and by induction on $t \in \mathcal{M}$ (left-to-right implication). The latter is presented below.

- $t = \lambda p.u$, with $u \in \mathcal{M}$. Then Φ cannot end with rule (abs) because σ is tight. The last rule of Φ is necessarily (abs_p) . The i.h. then applies and gives $b = e = m = 0$ and $f - 1 = |u|$. We conclude since $f = |u| + 1 = |t|$.
- $t = \langle t_1, t_2 \rangle$. Then Φ necessarily ends with rule (pair_p) and the counters are as required.
- $t = u[p \setminus v]$, with $u \in \mathcal{M}, v \in \mathcal{N}$. Then Φ ends with rule (match) , so that u (resp. v) is typable with some context Γ_u (resp. Γ_v), where $\Gamma = (\Gamma_u \parallel \text{var}(p)) \wedge \Gamma_v$. Let us consider the type \mathcal{A} of u in the premise of rule (match) . Since Γ_v is tight and $v \in \mathcal{N}$, then Lem. 12 guarantees that every type of v in \mathcal{A} is tight, and every counter typing v is of the form $(0, 0, 0, |v|)$. This same multitype \mathcal{A} types the pattern p , so that there are in principle two cases:
 - Either p is a variable typable with rule (pat_v) , but then $t \notin \mathcal{M}$ since t is still reducible. Contradiction.
 - Or p is typable with rule (pat_p) , so that its counter is $(0, 0, 1)$, its type is $[\bullet_{\mathcal{N}}]$ and its context is $\Gamma_u|_p$ necessarily tight by definition of rule (pat_p) .

Since $\Gamma_u \parallel \text{var}(p)$ is tight by hypothesis, then the whole context Γ_u is tight. We can then apply the i.h. to u and obtain counters for u of the form $b_u = e_u = m_u = 0$ and $f_u = |u|$. On the other side, since the type of p is $[\bullet_{\mathcal{N}}]$ (rule pat_p), there is only one premise to type v , which is necessarily of the form $\Delta \vdash^{(0,0,0,|v|)} v : \bullet_{\mathcal{N}}$. We then conclude that the counters typing $u[p \setminus v]$ are $b = e = m = 0$ and $f = f_u + f_v + 1 = |u| + |v| + 1 = |t|$, as required.

- $t \in \mathcal{N}$. We have three different cases.
 - $t = x$. This case is straightforward.

- $t = uv$, with $u \in \mathcal{N}$. Since Φ is tight, then Γ is tight and we can apply Lem. 12. Then Φ necessarily ends with rule (**app**). The i.h. then applies to the premise typing u , thus giving counters $b = e = m = 0$ and $f - 1 = |u|$. We conclude since $f = |u| + 1 = |t|$.
- $t = u[p \setminus v]$, with $u \in \mathcal{N}, v \in \mathcal{N}$. This case is similar to the third case. \blacktriangleleft

► **Lemma 14** (Substitution for System \mathcal{E}). *If $\Phi_t \triangleright \Gamma; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma$, and $\Phi_u \triangleright \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, then there exists $\Phi_{t\{x \setminus u\}} \triangleright \Gamma \wedge \Delta \vdash^{(b_t + b_u, e_t + e_u, m_t + m_u, f_t + f_u)} t\{x \setminus u\} : \sigma$.*

Proof. We generalise the statement as follows: Let $\Phi_u \triangleright \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$.

- If $\Phi_t \triangleright \Gamma; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma$, then there exists $\Phi_{t\{x \setminus u\}} \triangleright \Gamma \wedge \Delta \vdash^{(b_t + b_u, e_t + e_u, m_t + m_u, f_t + f_u)} t\{x \setminus u\} : \sigma$.
- If $\Phi_t \triangleright \Gamma; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \mathcal{B}$, then there exists $\Phi_{t\{x \setminus u\}} \triangleright \Gamma \wedge \Delta \vdash^{(b_t + b_u, e_t + e_u, m_t + m_u, f_t + f_u)} t\{x \setminus u\} : \mathcal{B}$.

The proof then follows by induction on Φ_t .

- If Φ_t is (**ax**), then we consider two cases:
 - $t = x$: then $\Phi_x \triangleright x : [\sigma'] \vdash^{(0,0,0,0)} x : \sigma'$ and $\Phi_u \triangleright \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : [\sigma']$, which is a consequence of $\Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \sigma'$. Then $x\{x \setminus u\} = u$, and we trivially obtain $\Phi_{t\{x \setminus u\}} \triangleright \Delta \vdash^{(0 + b_u, 0 + e_u, 0 + m_u, 0 + f_u)} u : \sigma'$.
 - $t = y$: then $\Phi_y \triangleright y : [\sigma]; x : [] \vdash^{(0,0,0,0)} y : \sigma$ and $\Phi_u \triangleright \emptyset \vdash^{(0,0,0,0)} u : []$. Then $y\{x \setminus u\} = y$, and we trivially obtain $\Phi_{t\{x \setminus u\}} \triangleright y : [\sigma] \vdash^{(0 + 0, 0 + 0, 0 + 0, 0 + 0)} y : \sigma$.
- If Φ_t ends with (**many**), then it has premises $(\Phi_t^i \triangleright \Gamma^i; x : \mathcal{A}^i \vdash^{(b_t^i, e_t^i, m_t^i, f_t^i)} t : \sigma^i)_{i \in I}$, where $\Gamma = \wedge_{i \in I} \Gamma^i$, $\mathcal{A} = \wedge_{i \in I} \mathcal{A}^i$, $b_t = +_{i \in I} b_t^i$, $e_t = +_{i \in I} e_t^i$, $m_t = +_{i \in I} m_t^i$, $f_t = +_{i \in I} f_t^i$ and $\mathcal{B} = [\sigma^i]_{i \in I}$. The derivation Φ_u can also be decomposed into several subderivations $(\Phi_u^i \triangleright \Delta^i \vdash^{(b_u^i, e_u^i, m_u^i, f_u^i)} u : \mathcal{A}^i)_{i \in I}$, where $b_u = +_{i \in I} b_u^i$, $e_u = +_{i \in I} e_u^i$, $m_u = +_{i \in I} m_u^i$, $f_u = +_{i \in I} f_u^i$, $\Delta = \wedge_{i \in I} \Delta^i$. We can apply the i.h. and we thus obtain derivations $(\Phi_{t\{x \setminus u\}}^i \triangleright \Gamma^i \wedge \Delta^i \vdash^{(b_t^i + b_u^i, e_t^i + e_u^i, m_t^i + m_u^i, f_t^i + f_u^i)} t\{x \setminus u\} : \sigma^i)_{i \in I}$. Then we apply rule (**many**) to get $\Phi_{t\{x \setminus u\}} \triangleright \Gamma \wedge \Delta \vdash^{(b_t + b_u, e_t + e_u, m_t + m_u, f_t + f_u)} t\{x \setminus u\} : \mathcal{B}$.
- If Φ_t ends with (**app**), so that $t = t'u'$, then

$$\Phi_{t'u'} \triangleright \Gamma_{t'} \wedge \Gamma_{u'}; x : \mathcal{A}_{t'} \wedge \mathcal{A}_{u'} \vdash^{(b_{t'} + b_{u'}, e_{t'} + e_{u'}, m_{t'} + m_{u'}, f_{t'} + f_{u'})} t'u' : \sigma,$$

which follows from the two term premises $\Gamma_{t'}; x : \mathcal{A}_{t'} \vdash^{(b_{t'}, e_{t'}, m_{t'}, f_{t'})} t' : \mathcal{B} \rightarrow \sigma$ and $\Gamma_{u'}; x : \mathcal{A}_{u'} \vdash^{(b_{u'}, e_{u'}, m_{u'}, f_{u'})} u' : \mathcal{B}$. Also, $\Phi_u \triangleright \Delta \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$ is a consequence of $(\Delta_k \vdash^{(b_u^k, e_u^k, m_u^k, f_u^k)} u : \sigma_k)_{k \in K}$, with $\mathcal{A} = [\sigma_k]_{k \in K}$, $\Delta = \wedge_{k \in K} \Delta_k$ and $b_u = +_{k \in K} b_u^k$, $e_u = +_{k \in K} e_u^k$, $m_u = +_{k \in K} m_u^k$ and $f_u = +_{k \in K} f_u^k$. Note on the other hand that $\mathcal{A} = \mathcal{A}_{t'} \wedge \mathcal{A}_{u'} = [\sigma_i]_{i \in K_{t'}} \wedge [\sigma_i]_{i \in K_{u'}}$, with $K = K_{t'} \uplus K_{u'}$, from which one can obtain (using the **many** rule):

- $\Delta_{t'} \vdash^{(B_{t'}, E_{t'}, M_{t'}, F_{t'})} u : \mathcal{A}_{t'}$
- $\Delta_{u'} \vdash^{(B_{u'}, E_{u'}, M_{u'}, F_{u'})} u : \mathcal{A}_{u'}$

where $b_u = B_{t'} + B_{u'} = (+_{i \in K_{t'}} b_u^i) + (+_{i \in K_{u'}} b_u^i)$, $e_u = E_{t'} + E_{u'} = (+_{i \in K_{t'}} e_u^i) + (+_{i \in K_{u'}} e_u^i)$, $m_u = M_{t'} + M_{u'} = (+_{i \in K_{t'}} m_u^i) + (+_{i \in K_{u'}} m_u^i)$, $f_u = F_{t'} + F_{u'} = (+_{i \in K_{t'}} f_u^i) + (+_{i \in K_{u'}} f_u^i)$. By the i.h. we have:

$$\Gamma_{t'} \wedge \Delta_{t'} \vdash^{(b_{t'} + B_{t'}, e_{t'} + E_{t'}, m_{t'} + M_{t'}, f_{t'} + F_{t'})} t'\{x \setminus u\} : \mathcal{B} \rightarrow \sigma$$

$$\Gamma_{u'} \wedge \Delta_{u'} \vdash^{(b_{u'} + B_{u'}, e_{u'} + E_{u'}, m_{u'} + M_{u'}, f_{u'} + F_{u'})} u'\{x \setminus u\} : \mathcal{B}$$

3:30 A Quantitative Understanding of Pattern Matching

Finally, applying the **app** rule we obtain:

$$\Gamma_{t'} \wedge \Gamma_{u'} \wedge \Delta_{t'} \wedge \Delta_{u'} \vdash^{(b,e,m,f)} (t'\{x\backslash u\})(u'\{x\backslash u\}) : \sigma$$

with $b = b_{t'} + b_{u'} + b_u$, $e = e_{t'} + e_{u'} + e_u$, $m = m_{t'} + m_{u'} + m_u$ and $f = f_{t'} + f_{u'} + f_u$, as expected.

- If Φ_t ends with **(abs)**, **(abs_p)**, **(app_p)** or **(match)** the result follows from the i.h. by assuming α -conversion whenever necessary.
- If Φ_t ends with **(pair)** or **(pair_p)**, so that $t = \langle t', u' \rangle$, then we have two cases. The case for **pair_p**, follows from Φ_t being of the form $x : [] \vdash^{(0,0,0,1)} \langle t', u' \rangle : \bullet_{\mathcal{M}}$, which implies $\Phi_{u \triangleright} \vdash^{(0,0,0,0)} u : []$. Therefore $\emptyset \vdash^{(0+0,0+0,0+0,1+0)} \langle t'\{x\backslash u\}, u'\{x\backslash u\} \rangle : \bullet_{\mathcal{M}}$ holds. The case for **pair** follows by induction following the same reasoning used in rule **app**. ◀

► **Lemma 15** (Exact Subject Reduction). *If $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t : \sigma$, and $t \rightarrow_{\mathfrak{h}} t'$ is an s -step, with $s \in \{b, e, m\}$, then $\Phi' \triangleright \Gamma \vdash^{(b',e',m',f)} t' : \sigma$, where*

- $s = b$ implies $b' = b - 1$, $e' = e$, $m' = m$.
- $s = e$ implies $b' = b$, $e' = e - 1$, $m' = m$.
- $s = m$ implies $b' = b$, $e' = e$, $m' = m - 1$.

Proof. By induction on $\rightarrow_{\mathfrak{h}}$.

- $t = \mathbb{L}[\lambda p.v]u \rightarrow_{\mathfrak{h}} \mathbb{L}[v[p\backslash u]] = t'$. The proof is by induction on the list context \mathbb{L} . We only show the case of the empty list as the other one is straightforward. The typing derivation Φ is necessarily of the form

$$\frac{\frac{\Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma \quad \Gamma_v|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A}}{\Gamma_v \Vdash \mathbf{var}(p) \vdash^{(b_v+1, e_v+e_p, m_v+m_p, f_v+f_p)} \lambda p.v : \mathcal{A} \rightarrow \sigma} \quad \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}}{\Gamma_v \Vdash \mathbf{var}(p) \wedge \Gamma_u \vdash^{(b_v+1+b_u, e_v+e_p+e_u, m_v+m_p+m_u, f_v+f_p+f_u)} (\lambda p.v)u : \sigma}$$

We then construct the following derivation Φ' :

$$\frac{\Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma \quad \Gamma_v|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A} \quad \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}}{\Gamma_v \Vdash \mathbf{var}(p) \wedge \Gamma_u \vdash^{(b_v+b_u, e_v+e_p+e_u, m_v+m_u+m_p, f_v+f_p+f_u)} v[p\backslash u] : \sigma}$$

The counters are as expected because the first one has decremented by 1.

- $t = v[x\backslash u] \rightarrow_{\mathfrak{h}} v\{x\backslash u\} = t'$, where $v \not\rightarrow_{\mathfrak{h}}$. Then Φ has two premises $\Gamma_v : x : \mathcal{A} \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma$ and $\Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, where $\Gamma = \Gamma_v \wedge \Gamma_u$, $b = b_v + b_u$, $e = e_v + e_u + 1$, $m = m_v + m_u + 0$, and $f = f_v + f_u + 0$.

Lem. 14 then gives a derivation ending with $\Gamma_v \wedge \Gamma_u \vdash^{(b_v+b_u, e_v+e_u, m_v+m_u, f_v+f_u)} v\{x\backslash u\} : \sigma$. The context, type, and counters are as expected.

- $t = v[\langle p_1, p_2 \rangle \backslash \mathbb{L}[\langle u_1, u_2 \rangle]] \rightarrow_{\mathfrak{h}} \mathbb{L}[v[p_1\backslash u_1][p_2\backslash u_2]] = t'$, where $v \not\rightarrow_{\mathfrak{h}}$.

Let $p = \langle p_1, p_2 \rangle$ and $u = \langle u_1, u_2 \rangle$. The typing derivation Φ is necessarily of the form

$$\frac{\frac{\Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma \quad \Gamma_v|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A} \quad \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} \mathbb{L}[u] : \mathcal{A}}{\Gamma_v \Vdash \mathbf{var}(p) \wedge \Gamma_u \vdash^{(b_v+b_u, e_v+e_u+e_p, m_v+m_u+m_p, f_v+f_u+f_p)} v[\langle p_1, p_2 \rangle \backslash \mathbb{L}[u]] : \sigma}}$$

Then $\mathcal{A} = [\times(\mathcal{A}_1, \mathcal{A}_2)]$, for some multitypes \mathcal{A}_1 and \mathcal{A}_2 , and so the pattern $\langle p_1, p_2 \rangle$ is typable as follows:

$$\frac{\Gamma_v|_{p_1} \Vdash^{(e_1, m_1, f_1)} p_1 : \mathcal{A}_1 \quad \Gamma_v|_{p_2} \Vdash^{(e_2, m_2, f_2)} p_2 : \mathcal{A}_2}{\Gamma_v|_p \Vdash^{(e_1+e_2, 1+m_1+m_2, f_1+f_2)} \langle p_1, p_2 \rangle : [\times(\mathcal{A}_1, \mathcal{A}_2)]}$$

where $e_p = e_1 + e_2$, $m_p = 1 + m_1 + m_2$ and $f_p = f_1 + f_2$. The proof then follows by induction on list \mathbb{L} :

- For $L = \square$ we have term u typable as follows:

$$\frac{\frac{\Gamma_1 \vdash (b'_1, e'_1, m'_1, f'_1) u_1 : \mathcal{A}_1 \quad \Gamma_2 \vdash (b'_2, e'_2, m'_2, f'_2) u_2 : \mathcal{A}_2}{\Gamma_u \vdash (b_u, e_u, m_u, f_u) \langle u_1, u_2 \rangle : \times(\mathcal{A}_1, \mathcal{A}_2)}}{\Gamma_u \vdash (b_u, e_u, m_u, f_u) \langle u_1, u_2 \rangle : \mathcal{A}}$$

where $\Gamma_u = \Gamma_1 \wedge \Gamma_2$ and $(b_u, e_u, m_u, f_u) = (b'_1 + b'_2, e'_1 + e'_2, m'_1 + m'_2, f'_1 + f'_2)$.
We first construct the following derivation:

$$\frac{\Gamma_v \vdash (b_v, e_v, m_v, f_v) v : \sigma \quad \Gamma_v|_{p_1} \Vdash (e_1, m_1, f_1) p_1 : \mathcal{A}_1 \quad \Gamma_1 \vdash (b'_1, e'_1, m'_1, f'_1) u_1 : \mathcal{A}_1}{\Gamma_v \parallel \mathbf{var}(p_1) \wedge \Gamma_1 \vdash (b_v + b'_1, e_v + e'_1 + e_1, m_v + m'_1 + m_1, f_v + f'_1 + f_1) v[p_1 \setminus u_1] : \sigma}$$

By using relevance and α -conversion to assume freshness of bound variables, we can construct a derivation with conclusion

$$\Gamma_v \parallel \mathbf{var}(p_1) \parallel \mathbf{var}(p_2) \wedge \Gamma_u \vdash (b', e', m', f) v[p_1 \setminus u_1][p_2 \setminus u_2] : \sigma$$

where $(b', e', m', f) = (b_v + b_u, e_v + e_u + e_1 + e_2, m_v + m_u + m_1 + m_2, f_v + f_u + f_1 + f_2)$.
In order to conclude we remark the following facts:

- * $\Gamma_v \parallel \mathbf{var}(\langle p_1, p_2 \rangle) = \Gamma_v \parallel \mathbf{var}(p_1) \parallel \mathbf{var}(p_2)$
- * $b_v + b_u = b_v + b'_1 + b'_2$
- * $e_v + e_u + e_p = e_v + e'_1 + e'_2 + e_1 + e_2$
- * $m_v + m_u + m_p = m_v + m'_1 + m'_2 + 1 + m_1 + m_2$
- * $f_v + f_u + f_p = f_v + f'_1 + f'_2 + f_1 + f_2$

Then the context, type and counters are as expected.

- Let $L = L'[q \setminus s]$. Then Φ_u is necessarily of the following form:

$$\frac{\frac{\Delta_u \vdash (b'_u, e'_u, m'_u, f'_u) L'[[u]] : \times(\mathcal{A}_1, \mathcal{A}_2) \quad \Delta_u|_q \Vdash (e_q, m_q, f_q) q : \mathcal{B} \quad \Delta_s \vdash (b_s, e_s, m_s, f_s) s : \mathcal{B}}{\Gamma_u \vdash (b'_u + b_s, e'_u + e_s + e_q, m'_u + m_s + m_q, f'_u + f_s + f_q) L'[[u]][q \setminus s] : \times(\mathcal{A}_1, \mathcal{A}_2)}}{\Gamma_u \vdash (b'_u + b_s, e'_u + e_s + e_q, m'_u + m_s + m_q, f'_u + f_s + f_q) L'[[u]][q \setminus s] : \mathcal{A}}$$

where $\Gamma_u = \Delta_u \parallel \mathbf{var}(q) \wedge \Delta_s$, $b_u = b'_u + b_s$, $e_u = e'_u + e_s + e_q$, $m_u = m'_u + m_s + m_q$ and $f_u = f'_u + f_s + f_q$.

We will apply the i.h. on the reduction step $v[p \setminus L'[[u]]] \rightarrow_p L'[[v[p_1 \setminus u_1][p_2 \setminus u_2]]]$, in particular we type the left-hand side term with the following derivation Ψ_1 :

$$\frac{\Gamma_v \vdash (b_v, e_v, m_v, f_v) v : \sigma \quad \Gamma_v|_p \Vdash (e_p, m_p, f_p) p : \mathcal{A} \quad \frac{\Delta_u \vdash (b'_u, e'_u, m'_u, f'_u) L'[[u]] : \times(\mathcal{A}_1, \mathcal{A}_2)}{\Delta_u \vdash (b'_u, e'_u, m'_u, f'_u) L'[[u]] : \mathcal{A}}}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Delta_u \vdash (b_v + b'_u, e_v + e'_u + e_p, m_v + m'_u + m_p, f_v + f'_u + f_p) v[p \setminus L'[[u]]] : \sigma}$$

where $b_1 = b_v + b'_u$, $e_1 = e_v + e'_u + e_p$, $m_1 = m_v + m'_u + m_p$ and $f_1 = f_v + f'_u + f_p$. The i.h. gives a derivation $\Psi_2 \triangleright \Gamma_v \parallel \mathbf{var}(p) \wedge \Delta_u \vdash (b_2, e_2, m_2, f_2) L'[[v[p_1 \setminus u_1][p_2 \setminus u_2]]] : \sigma$ where $b_2 = b_1$, $e_2 = e_1$, $m_2 = m_1 - 1$ and $f_2 = f_1$. Let $\Lambda = \Gamma_v \parallel \mathbf{var}(p) \wedge \Delta_u$. We conclude with the following derivation Φ' :

$$\frac{\Psi_2 \quad \Delta_u|_q \Vdash (e_q, m_q, f_q) q : \mathcal{B} \quad \Delta_s \vdash (b_s, e_s, m_s, f_s) s : \mathcal{B}}{\Lambda \parallel \mathbf{var}(q) \wedge \Delta_s \vdash (b_2 + b_s, e_2 + e_s + e_q, m_2 + m_s + m_q, f_2 + f_s + f_q) L'[[v[p_1 \setminus u_1][p_2 \setminus u_2]]][q \setminus s] : \sigma}$$

Indeed, we first remark that $\Lambda|_q = \Delta_u|_q$ holds by relevance and α -conversion. Secondly, $\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u = \Gamma_v \parallel \mathbf{var}(p) \wedge (\Delta_u \parallel \mathbf{var}(q)) \wedge \Delta_s = \Lambda \parallel \mathbf{var}(q) \wedge \Delta_s$ also holds by relevance and α -conversion. Last, we conclude with the following remarks:

3:32 A Quantitative Understanding of Pattern Matching

- * $b' = b_2 + b_s = b_1 + b_s = b_v + b_u = b$
- * $e' = e_2 + e_s + e_q = e_1 + e_s + e_q = e_v + e_u + e_p = e$
- * $m' = m_2 + m_s + m_q = m_1 - 1 + m_s + m_q = m_v + m_u + m_p - 1 = m - 1$
- * $f' = f_2 + f_s + f_q = f_1 + f_s + f_q = f_v + f_u + f_p = f$

- Most of the inductive cases are straightforward, so we only show the interesting one. Let $t = v[p \setminus u] \rightarrow_h v[p \setminus u'] = t'$, where $v \not\rightarrow_h$ and $p \neq x$ and $u \rightarrow_h u'$. By construction there are typing subderivations $\Phi_v \triangleright \Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma$, $\Phi_p \triangleright \Gamma_v|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A}$ and $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$. Since p is not a variable then Φ_p ends with rule pat_p or pat_x . In both cases \mathcal{A} contains only one type, let us say $\mathcal{A} = [\sigma_u]$. Then Φ_u has the following form

$$\frac{\Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \sigma_u}{\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : [\sigma_u]}$$

The i.h. applied to the premise of Φ_u gives a derivation $\Gamma_u \vdash^{(b'_u, e'_u, m'_u, f_u)} u' : \sigma_u$ and having the expected counters. To conclude we build a type derivation Φ' for $v[p \setminus u']$ having the expected counters. ◀

C Completeness for System \mathcal{E}

- **Lemma 17** (Canonical Forms and Tight Derivations). *Let $t \in \mathcal{M}$. There exists a tight derivation $\Phi \triangleright \Gamma \vdash^{(0,0,0,|t|)} t : \mathfrak{t}$.*

Proof. We generalise the property to the two following statements, proved by structural induction on $t \in \mathcal{N}$, $t \in \mathcal{M}$, respectively, using relevance (Lem. 8).

- If $t \in \mathcal{N}$, then there exists a tight derivation $\Phi \triangleright \Gamma \vdash^{(0,0,0,|t|)} t : \bullet_{\mathcal{N}}$:
 - If $t = x$, then $x : [\bullet_{\mathcal{N}}] \vdash^{(0,0,0,0)} x : \bullet_{\mathcal{N}}$ by (ax), where $|x| = 0$.
 - If $t = uv$ where $u \in \mathcal{N}$, then $|t| = |u| + 1$ and by i.h. there is a tight derivation $\Phi_u \triangleright \Gamma_u \vdash^{(0,0,0,|u|)} u : \bullet_{\mathcal{N}}$. Then

$$\frac{\Gamma_u \vdash^{(0,0,0,|u|)} u : \bullet_{\mathcal{N}}}{\Gamma_u \vdash^{(0,0,0,|u|+1)} uv : \bullet_{\mathcal{N}}}$$

The result then holds for $\Gamma := \Gamma_u$.

- If $t = u[\langle p_1, p_2 \rangle \setminus v]$ where $u, v \in \mathcal{N}$, then $|t| = |u| + |v| + 1$ and by i.h. there are tight derivations $\Phi_u \triangleright \Gamma_u \vdash^{(0,0,0,|u|)} u : \bullet_{\mathcal{N}}$, $\Phi_v \triangleright \Gamma_v \vdash^{(0,0,0,|v|)} v : \bullet_{\mathcal{N}}$. Then, $\Phi'_v \triangleright \Gamma_v \vdash^{(0,0,0,|v|)} v : [\bullet_{\mathcal{N}}]$ and

$$\frac{\Phi_u \quad \Gamma_u|_{\langle p_1, p_2 \rangle} \Vdash^{(0,0,1)} \langle p_1, p_2 \rangle : [\bullet_{\mathcal{N}}] \quad \Phi'_v}{(\Gamma_u \parallel \text{var}(\langle p_1, p_2 \rangle)) \wedge \Gamma_v \vdash^{(0,0,0,|u|+|v|+1)} u[\langle p_1, p_2 \rangle \setminus v] : \bullet_{\mathcal{N}}}$$

The result then holds for $\Gamma := (\Gamma_u \parallel \text{var}(\langle p_1, p_2 \rangle)) \wedge \Gamma_v$, since by i.h. $\text{tight}(\Gamma_u)$ and $\text{tight}(\Gamma_v)$ thus $\text{tight}(\Gamma)$.

- If $t \in \mathcal{M}$, then there exists a tight derivation $\Phi \triangleright \Gamma \vdash^{(0,0,0,|t|)} t : \mathfrak{t}$.
 - If $t \in \mathcal{N}$ then by the previous item the result holds for $\mathfrak{t} := \bullet_{\mathcal{N}}$.
 - If $t = \langle u, v \rangle$ then $|t| = 1$ and $\vdash^{(0,0,0,1)} \langle u, v \rangle : \bullet_{\mathcal{M}}$ by (pair_p). The result then holds for $\Gamma := \emptyset$.

- If $t = \lambda p.u$ where $u \in \mathcal{M}$ then $|t| = |u| + 1$ and, by i.h., there is a tight derivation $\Phi_u \triangleright \Gamma_u \vdash^{(0,0,0,|u|)} u : \mathbf{t}$. Then

$$\frac{\Phi_u \triangleright \Gamma_u \vdash^{(0,0,0,|u|)} u : \mathbf{t}}{\Gamma_u \parallel \mathbf{var}(p) \vdash^{(0,0,0,|u|+1)} \lambda p.u : \bullet \mathcal{M}}$$

The result then holds for $\Gamma := \Gamma_u \parallel \mathbf{var}(p)$. Observe that, since by i.h. $\mathbf{tight}(\Gamma_u)$ holds, and $\Gamma_u \parallel \mathbf{var}(p) \subseteq \Gamma_u$ then $\mathbf{tight}(\Gamma_u \parallel \mathbf{var}(p))$ trivially holds.

- If $t = u[\langle p_1, p_2 \rangle \setminus v]$ where $u \in \mathcal{M}$ and $v \in \mathcal{N}$ then $|t| = |u| + |v| + 1$. Moreover, by the previous item there is a tight $\Phi_v \triangleright \Gamma_v \vdash^{(0,0,0,|v|)} v : \bullet \mathcal{N}$ (so that $\mathbf{tight}(\Gamma_v)$) and, by i.h., there is a tight derivation $\Phi_u \triangleright \Gamma_u \vdash^{(0,0,0,|u|)} u : \mathbf{t}$. Then, $\Phi'_v \triangleright \Gamma_v \vdash^{(0,0,0,|v|)} v : [\bullet \mathcal{N}]$ and

$$\frac{\Phi_u \quad (\Gamma_u)_{\langle p, q \rangle} \Vdash^{(0,0,1)} \langle p, q \rangle : [\bullet \mathcal{N}] \quad \Phi'_v}{(\Gamma_u \parallel \mathbf{var}(\langle p_1, p_2 \rangle)) \wedge \Gamma_v \vdash^{(0,0,0,|u|+|v|+1)} u[\langle p_1, p_2 \rangle \setminus v] : \mathbf{t}}$$

The result then holds for $\Gamma := (\Gamma_u \parallel \mathbf{var}(\langle p_1, p_2 \rangle)) \wedge \Gamma_v$, since $\mathbf{tight}(\Gamma_v)$ as remarked, and by i.h. $\mathbf{tight}(\Gamma_u)$, thus $\mathbf{tight}(\Gamma)$. \blacktriangleleft

► **Lemma 18** (Anti-Substitution for System \mathcal{E}). *Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t\{x \setminus u\} : \sigma$. Then, there exist derivations Φ_t, Φ_u , integers $b_t, b_u, e_t, e_u, m_t, m_u, f_t, f_u$, contexts Γ_t, Γ_u , and multitype \mathcal{A} such that $\Phi_t \triangleright \Gamma_t; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, $b = b_t + b_u$, $e = e_t + e_u$, $m = m_t + m_u$, $f = f_t + f_u$, and $\Gamma = \Gamma_t \wedge \Gamma_u$.*

Proof. As in the case of the substitution lemma, the proof follows by generalising the property for the two cases where the type derivation Φ assigns a type or a multiset type.

- Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t\{x \setminus u\} : \sigma$. Then, there exist derivations Φ_t, Φ_u , integers $b_t, b_u, e_t, e_u, m_t, m_u, f_t, f_u$, contexts Γ_t, Γ_u , and multitype \mathcal{A} such that $\Phi_t \triangleright \Gamma_t; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, $b = b_t + b_u$, $e = e_t + e_u$, $m = m_t + m_u$, $f = f_t + f_u$, and $\Gamma = \Gamma_t \wedge \Gamma_u$.
- Let $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} t\{x \setminus u\} : \mathcal{B}$. Then, there exist derivations Φ_t, Φ_u , integers $b_t, b_u, e_t, e_u, m_t, m_u, f_t, f_u$, contexts Γ_t, Γ_u , and multitype \mathcal{A} such that $\Phi_t \triangleright \Gamma_t; x : \mathcal{A} \vdash^{(b_t, e_t, m_t, f_t)} t : \mathcal{B}$, $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, $b = b_t + b_u$, $e = e_t + e_u$, $m = m_t + m_u$, $f = f_t + f_u$, and $\Gamma = \Gamma_t \wedge \Gamma_u$.

We will reason by induction on Φ . For all the rules (except **many**), we will have the trivial case $t\{x \setminus u\}$, where $t = x$, in which case $t\{x \setminus u\} = u$, for which we have a derivation $\Phi \triangleright \Gamma \vdash^{(b,e,m,f)} u : \sigma$. Therefore $\Phi_t \triangleright x : [\sigma] \vdash^{(0,0,0,0)} x : \sigma$ and $\Phi_u \triangleright \Gamma \vdash^{(b,e,m,f)} u : [\sigma]$ is obtained from Φ using the (**many**) rule. The conditions on the counters hold trivially. We now reason on the different cases assuming that $t \neq x$.

- If Φ is (**ax**), therefore $\Phi \triangleright y : [\sigma] \vdash^{(0,0,0,0)} y : \sigma$. We only consider the case where $t = y$ and $y \neq x$. Then we take $\mathcal{A} = []$, $\Phi_t \triangleright y : [\sigma]; x : [] \vdash^{(0,0,0,0)} y : \sigma$ and $\Phi_u \triangleright \vdash^{(0,0,0,0)} u : []$, using rule (**many**). The conditions on the counters follow trivially.
- If Φ ends with (**many**), then $\Phi \triangleright \bigwedge_{k \in K} \Gamma_k \vdash^{(+_{k \in K} b_k, +_{k \in K} e_k, +_{k \in K} m_k, +_{k \in K} f_k)} t\{x \setminus u\} : [\sigma_k]_{k \in K}$ follows from $\Phi^k \triangleright \Gamma_k \vdash^{(b_k, e_k, m_k, f_k)} t\{x \setminus u\} : \sigma_k$, for each $k \in K$. By the i.h. there exist $\Phi_t^k, \Phi_u^k, b_t^k, b_u^k, e_t^k, e_u^k, m_t^k, m_u^k, f_t^k, f_u^k$, contexts Γ_t^k, Γ_u^k and multitype \mathcal{A}_k , such that $\Phi_t^k \triangleright \Gamma_t^k; x : \mathcal{A}_k \vdash^{(b_t^k, e_t^k, m_t^k, f_t^k)} t : \sigma_k$, $\Phi_u^k \triangleright \Gamma_u^k \vdash^{(b_u^k, e_u^k, m_u^k, f_u^k)} u : \mathcal{A}_k$, $\Gamma_k = \Gamma_t^k \wedge \Gamma_u^k$, $b_k = b_t^k + b_u^k$, $e_k = e_t^k + e_u^k$, $m_k = m_t^k + m_u^k$, $f_k = f_t^k + f_u^k$.

Taking $\mathcal{A} = \bigwedge_{k \in K} \mathcal{A}_k$ and using the (**many**) rule on the derivations $(\Phi_t^k)_{k \in K}$ we get now $\bigwedge_{k \in K} \Gamma_t^k; x : \mathcal{A} \vdash^{(+_{k \in K} b_t^k, +_{k \in K} e_t^k, +_{k \in K} m_t^k, +_{k \in K} f_t^k)} t : [\sigma_k]_{k \in K}$. From the premises $(\Phi_u^k)_{k \in K}$, by applying again the (**many**) rule, we get $\bigwedge_{k \in K} \Gamma_u^k \vdash^{(+_{k \in K} b_u^k, +_{k \in K} e_u^k, +_{k \in K} m_u^k, +_{k \in K} f_u^k)} u : \mathcal{A}$.

3:34 A Quantitative Understanding of Pattern Matching

Note that $\Gamma = \bigwedge_{k \in K} \Gamma_k = (\bigwedge_{k \in K} \Gamma_t^k) \wedge (\bigwedge_{k \in K} \Gamma_u^k)$, $b = +_{k \in K} b_k = (+_{k \in K} b_t^k) + (+_{k \in K} b_u^k)$, $e = +_{k \in K} e_k = (+_{k \in K} e_t^k) + (+_{k \in K} e_u^k)$, $m = +_{k \in K} m_k = (+_{k \in K} m_t^k) + (+_{k \in K} m_u^k)$ and $f = +_{k \in K} f_k = (+_{k \in K} f_t^k) + (+_{k \in K} f_u^k)$, as expected.

- If Φ ends with **(abs)**, then $t = \lambda p.t'$, therefore

$$\Phi \triangleright \Gamma \parallel \mathbf{var}(p) \vdash^{(b_t+1, e_t+e_p, m_t+m_p, f_t+f_p)} \lambda p.(t'\{x \setminus u\}) : \mathcal{B} \rightarrow \sigma$$

follows from $\Gamma \vdash^{(b_t, e_t, m_t, f_t)} t'\{x \setminus u\} : \sigma$ and $\Gamma|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{B}$. On the other side one can always assume that $\mathbf{var}(p) \cap \mathbf{fv}(u) = \emptyset$ and $x \notin \mathbf{var}(p)$. We can then apply the i.h. to obtain $\Phi_{t'} \triangleright \Gamma_{t'}; x : \mathcal{A} \vdash^{(b_{t'}, e_{t'}, m_{t'}, f_{t'})} t' : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, with $\Gamma = \Gamma_{t'} \wedge \Gamma_u$, $b_t = b_{t'} + b_u$, $e_t = e_{t'} + e_u$, $m_t = m_{t'} + m_u$, $f_t = f_{t'} + f_u$. Then using rule **(abs)** we get $\Phi_t \triangleright \Gamma_{t'} \parallel \mathbf{var}(p); x : \mathcal{A} \vdash^{(b_{t'}+1, e_{t'}+e_p, m_{t'}+m_p, f_{t'}+f_p)} \lambda p.t' : \mathcal{B} \rightarrow \sigma$. And $\Gamma \parallel \mathbf{var}(p) = (\Gamma_{t'} \parallel \mathbf{var}(p)) \wedge \Gamma_u$, $b_t + 1 = b_{t'} + 1 + b_u$, $e_t = e_{t'} + e_u$, $m_t = m_{t'} + m_u$ and $f_t = f_{t'} + f_u$, as expected.

- If Φ ends with **(app)** then $t = t'u'$, and the derivation

$$\Phi \triangleright \Gamma \wedge \Delta \vdash^{(b_{t'}+b_{u'}, e_{t'}+e_{u'}, m_{t'}+m_{u'}, f_{t'}+f_{u'})} t'\{x \setminus u\} u'\{x \setminus u\} : \sigma$$

follows from $\Gamma \vdash^{(b_{t'}, e_{t'}, m_{t'}, f_{t'})} t'\{x \setminus u\} : \mathcal{B} \rightarrow \sigma$ and $\Delta \vdash^{(b_{u'}, e_{u'}, m_{u'}, f_{u'})} u'\{x \setminus u\} : \mathcal{B}$. By the i.h. there exist $\Phi_{t'}, \Phi_{t'}^u, b_{t''}, b_{t''}^u, e_{t''}, e_{t''}^u, m_{t''}, m_{t''}^u, f_{t''}, f_{t''}^u$, contexts $\Gamma_{t''}, \Gamma_{t''}^u$ and multitype $\mathcal{A}_{t''}$, such that

$$\Phi_{t'} \triangleright \Gamma_{t''}; x : \mathcal{A}_{t''} \vdash^{(b_{t''}, e_{t''}, m_{t''}, f_{t''})} t' : \mathcal{B} \rightarrow \sigma \quad \Phi_{t'}^u \triangleright \Gamma_{t''}^u \vdash^{(b_{t''}^u, e_{t''}^u, m_{t''}^u, f_{t''}^u)} u : \mathcal{A}_{t''}$$

where $b_{t'} = b_{t''} + b_{t''}^u$, $e_{t'} = e_{t''} + e_{t''}^u$, $m_{t'} = m_{t''} + m_{t''}^u$, $f_{t'} = f_{t''} + f_{t''}^u$ and $\Gamma = \Gamma_{t''} \wedge \Gamma_{t''}^u$. And by the i.h. applied to the second premise of Φ , there exist $\Phi_{u'}, \Phi_{u'}^u, b_{u''}, b_{u''}^u, e_{u''}, e_{u''}^u, m_{u''}, m_{u''}^u, f_{u''}, f_{u''}^u$, contexts $\Gamma_{u''}, \Gamma_{u''}^u$ and multitype $\mathcal{A}_{u''}$, such that

$$\Phi_{u'} \triangleright \Delta_{u''}; x : \mathcal{A}_{u''} \vdash^{(b_{u''}, e_{u''}, m_{u''}, f_{u''})} u' : \mathcal{B} \quad \Phi_{u'}^u \triangleright \Delta_{u''}^u \vdash^{(b_{u''}^u, e_{u''}^u, m_{u''}^u, f_{u''}^u)} u : \mathcal{A}_{u''}$$

where $b_{u'} = b_{u''} + b_{u''}^u$, $e_{u'} = e_{u''} + e_{u''}^u$, $m_{u'} = m_{u''} + m_{u''}^u$, $f_{u'} = f_{u''} + f_{u''}^u$ and $\Delta = \Delta_{u''} \wedge \Delta_{u''}^u$. Now, taking $\mathcal{A} = \mathcal{A}_{t''} \wedge \mathcal{A}_{u''}$, and using the **(app)** rule, one gets a derivation of the form $\Phi_{t'u'} \triangleright \Gamma_{t''} \wedge \Delta_{u''}; x : \mathcal{A}_{t''} \wedge \mathcal{A}_{u''} \vdash^{(b_{t''}+b_{u''}, e_{t''}+e_{u''}, m_{t''}+m_{u''}, f_{t''}+f_{u''})} t' : \mathcal{B} \rightarrow \sigma$ and applying the **(many)** rule to the premises of $\Phi_{t''}^u$ and $\Phi_{u''}^u$ one gets a derivation of the form $\Phi_u \triangleright \Gamma_{t''}^u \wedge \Delta_{u''}^u \vdash^{(b_{t''}^u+b_{u''}^u, e_{t''}^u+e_{u''}^u, m_{t''}^u+m_{u''}^u, f_{t''}^u+f_{u''}^u)} u : \mathcal{A}$. Note that $\Gamma \wedge \Delta = (\Gamma_{t''} \wedge \Gamma_{t''}^u) \wedge (\Delta_{u''} \wedge \Delta_{u''}^u) = (\Gamma_{t''} \wedge \Delta_{u''}) \wedge (\Gamma_{t''}^u \wedge \Delta_{u''}^u)$ and $b = b_{t'} + b_{u'} = (b_{t''} + b_{t''}^u) + (b_{u''} + b_{u''}^u) = (b_{t''} + b_{u''}) + (b_{t''}^u + b_{u''}^u)$ as expected (the same happens for the remaining counters).

- If Φ ends with **(abs_p)**, **(app_p)** or **(match)** the result follows from the inductive hypothesis, as in the previous cases.
- If Φ ends with **(pair)** or **(pair_p)**, so that $t = \langle t', u' \rangle$, then we have two cases. The case for **pair_p**, follows from Φ being of the form $\vdash^{(0,0,0,1)} \langle t'\{x \setminus u\}, u'\{x \setminus u\} \rangle : \bullet_{\mathcal{M}}$. We then take $\mathcal{A} = []$, $\Phi_{\langle t', u' \rangle} \triangleright x : [] \vdash^{(0,0,0,1)} \langle t', u' \rangle : \bullet_{\mathcal{M}}$ and $\Phi_u \triangleright \vdash^{(0,0,0,0)} u : []$ follows trivially from the **(many)** rule. Then conditions on counters and contexts hold trivially. The case for **(pair)** follows by induction using the same reasoning as in rule **(app)**. ◀

► **Lemma 19** (Exact Subject Expansion). *If $\Phi \triangleright \Gamma \vdash^{(b', e', m', f')} t' : \sigma$, and $t \rightarrow_{\mathfrak{h}} t'$ is an s -step, with $s \in \{b, e, m\}$, then $\Phi \triangleright \Gamma \vdash^{(b, e, m, f)} t : \sigma$, where*

- $s = b$ implies $b = b' + 1$, $e' = e$, $m' = m$.
- $s = e$ implies $b' = b$, $e = e' + 1$, $m' = m$.
- $s = m$ implies $b' = b$, $e' = e$, $m = m' + 1$.

Proof. By induction on \rightarrow_h , using Lem. 18.

- $t = \mathbb{L}[\lambda p.v]u \rightarrow_h \mathbb{L}[v[p \setminus u]] = t'$. The proof is by induction on the list context \mathbb{L} .

If $\mathbb{L} = \square$ then by construction there are derivations $\Phi_v \triangleright \Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma$ and $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$ for some multitype \mathcal{A} , and Φ' is of the form

$$\frac{\Phi_v \quad (\Gamma_v)|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A} \quad \Phi_u}{(\Gamma_v \parallel \mathbf{var}(p)) \wedge \Gamma_u \vdash^{(b', e', m', f)} v[p \setminus u] : \sigma}$$

where $\Gamma = (\Gamma_v \parallel \mathbf{var}(p)) \wedge \Gamma_u$, $b' = b_v + b_u$, $e' = e_v + e_u + e_p$, $m' = m_v + m_u + m_p$ and $f = f_v + f_u + f_p$. Then

$$\Phi \triangleright \frac{\frac{\Phi_v \quad (\Gamma_v)|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A}}{\Gamma_v \parallel \mathbf{var}(p) \vdash^{(b_v+1, e_v+e_p, m_v+m_p, f_v+f_p)} \lambda p.v : \mathcal{A} \rightarrow \sigma} \quad \Phi_u}{(\Gamma_v \parallel \mathbf{var}(p)) \wedge \Gamma_u \vdash^{((b_v+1)+b_u, (e_v+e_p)+e_u, (m_v+m_p)+m_u, (f_v+f_p)+f_u)} (\lambda p.v)u : \sigma}$$

where $b = (b_v + 1) + b_u = b' + 1$, $e = (e_v + e_p) + e_u = e'$ and $m = (m_v + m_p) + m_u = m'$. If $\mathbb{L} \neq \square$ the proof from the i.h. is straightforward.

- $t = v[x \setminus u] \rightarrow_h v\{x \setminus u\} = t'$, where $v \not\rightarrow_h$. Then by the anti-substitution property (Lem. 18) there exist derivations Φ_v, Φ_u , integers $b_v, b_u, e_v, e_u, m_v, m_u, f_v, f_u$, contexts Γ_v, Γ_u , and multitype \mathcal{A} such that $\Phi_v \triangleright \Gamma_v; x : \mathcal{A} \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma$, $\Phi_u \triangleright \Gamma_u \vdash^{(b_u, e_u, m_u, f_u)} u : \mathcal{A}$, $b' = b_v + b_u$, $e' = e_v + e_u$, $m' = m_v + m_u$, $f = f_v + f_u$, and $\Gamma = \Gamma_v \wedge \Gamma_u$. Then,

$$\Phi \triangleright \frac{\Phi_v \quad (\Gamma_v; x : \mathcal{A})|_x \Vdash^{(1, 0, 0)} x : \mathcal{A} \quad \Phi_u}{\Gamma_v \wedge \Gamma_u \vdash^{(b_v+b_u, e_v+e_u+1, m_v+m_u+0, f_v+f_u+0)} v[x \setminus u] : \sigma}$$

where $b = b_v + b_u = b'$, $e = e_v + e_u + 1 = e' + 1$ and $m = m_v + m_u = m'$. Note that $(\Gamma_v; x : \mathcal{A}) \parallel \mathbf{var}(x) = \Gamma_v$.

- $t = v[\langle p_1, p_2 \rangle \setminus \mathbb{L}[\langle u_1, u_2 \rangle]] \rightarrow_h \mathbb{L}[v[p_1 \setminus u_1][p_2 \setminus u_2]] = t'$, where $v \not\rightarrow_h$. Let us abbreviate $p = \langle p_1, p_2 \rangle$ and $u = \langle u_1, u_2 \rangle$. The proof is by induction on the list \mathbb{L} .
 - $\mathbb{L} = \square$, then there are $\Phi_v \triangleright \Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma$, $\Phi_1 \triangleright \Gamma_1 \vdash^{(b_u^1, e_u^1, m_u^1, f_u^1)} u_1 : \mathcal{A}_1$ and $\Phi_2 \triangleright \Gamma_2 \vdash^{(b_u^2, e_u^2, m_u^2, f_u^2)} u_2 : \mathcal{A}_2$ where

$$\Phi_{v[p_1 \setminus u_1]} \triangleright \frac{\Phi_v \quad \Gamma_v|_{p_1} \Vdash^{(e_1, m_1, f_1)} p_1 : \mathcal{A}_1 \quad \Phi_1}{(\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1 \vdash^{(b_v+b_u^1, e_v+e_u^1+e_1, m_v+m_u^1+m_1, f_v+f_u^1+f_1)} v[p_1 \setminus u_1] : \sigma}$$

and

$$\Phi' \triangleright \frac{\Phi_{v[p_1 \setminus u_1]} \quad ((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1)|_{p_2} \Vdash^{(e_2, m_2, f_2)} p_2 : \mathcal{A}_2 \quad \Phi_2}{\Gamma \vdash^{(b', e', m', f)} v[p_1 \setminus u_1][p_2 \setminus u_2] : \sigma}$$

where $b' = b_v +_{i=1,2} b_u^i$, $e' = e_v +_{i=1,2} e_u^i + e_i$, $m' = m_v +_{i=1,2} m_u^i + m_i$, $f = f_v +_{i=1,2} f_u^i + f_i$ and $\Gamma = (((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1) \parallel \mathbf{var}(p_2)) \wedge \Gamma_2$.

Moreover, $((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1) \parallel \mathbf{var}(p_2) = ((\Gamma_v \parallel \mathbf{var}(p_1)) \parallel \mathbf{var}(p_2)) \wedge \Gamma_1 \parallel \mathbf{var}(p_2)$, where $(\Gamma_v \parallel \mathbf{var}(p_1)) \parallel \mathbf{var}(p_2) = \Gamma_v \parallel \mathbf{var}(\langle p_1, p_2 \rangle)$ and $\Gamma_1 \parallel \mathbf{var}(p_2) =_{L. 8} \Gamma_1$. Similarly, $((\Gamma_v \parallel \mathbf{var}(p_1)) \wedge \Gamma_1)|_{p_2} =_{Lem. 8} (\Gamma_v \parallel \mathbf{var}(p_1))|_{p_2}$ and, by linearity of patterns, $(\Gamma_v \parallel \mathbf{var}(p_1))|_{p_2} = \Gamma_v|_{p_2}$. Hence,

$$\Phi_{\langle p_1, p_2 \rangle} \triangleright \frac{\Gamma_v|_{p_1} \Vdash^{(e_1, m_1, f_1)} p_1 : \mathcal{A}_1 \quad \Gamma_v|_{p_2} \Vdash^{(e_2, m_2, f_2)} p_2 : \mathcal{A}_2}{\Gamma_v|_{p_1} \wedge \Gamma_v|_{p_2} \Vdash^{(e_1+e_2, 1+m_1+m_2, f_1+f_2)} \langle p_1, p_2 \rangle : [\times(\mathcal{A}_1, \mathcal{A}_2)]}$$

where $\Gamma_v|_{p_1} \wedge \Gamma_v|_{p_2} = \Gamma_v|_{\langle p_1, p_2 \rangle}$, and $\Phi_{\langle u_1, u_2 \rangle} \triangleright \Gamma_1 \wedge \Gamma_2 \vdash^{(b_u^1+b_u^2, e_u^1+e_u^2, m_u^1+m_u^2, f_u^1+f_u^2)} \langle u_1, u_2 \rangle : [\times(\mathcal{A}_1, \mathcal{A}_2)]$. Therefore,

$$\Phi \triangleright \frac{\Phi_v \quad \Phi_{\langle p_1, p_2 \rangle} \quad \Phi_{\langle u_1, u_2 \rangle}}{(\Gamma_v \parallel \mathbf{var}(\langle p_1, p_2 \rangle)) \wedge (\Gamma_1 \wedge \Gamma_2) \vdash^{(b, e, m, f)} v[\langle p_1, p_2 \rangle \setminus \langle u_1, u_2 \rangle] : \sigma}$$

where $b = b_v +_{i=1,2} b_u^i = b'$, $e' = e_v +_{i=1,2} e_u^i + e_i = e'$ and $m = 1 + m_v +_{i=1,2} m_u^i + m_i = m' + 1$.

- If $L = L'[q \setminus s]$, then Φ' is of the form:

$$\frac{\Gamma_{L'} \vdash^{(b'_i, e'_i, m'_i, f'_i)} L' \llbracket v[p_1 \setminus u_1][p_2 \setminus u_2] \rrbracket : \sigma \quad \Gamma_{L'}|_q \Vdash^{(e_q, m_q, f_q)} q : \mathcal{A} \quad \Gamma_s \vdash^{(b_s, e_s, m_s, f_s)} s : \mathcal{A}}{\Gamma_{L'} \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash^{(b'_l + b_s, e'_l + e_s + e_q, m'_l + m_s + m_q, f'_l + f_s + f_q)} L' \llbracket v[p_1 \setminus u_1][p_2 \setminus u_2] \rrbracket [q \setminus s] : \sigma}$$

where $b' = b'_l + b_s$, $e' = e'_l + e_s + e_q$, $m' = m'_l + m_s + m_q$, $f' = f'_l + f_s + f_q$ and $\Gamma = \Gamma_{L'} \parallel \mathbf{var}(q) \wedge \Gamma_s$. From $v[\langle p_1, p_2 \rangle \setminus L' \llbracket \langle u_1, u_2 \rangle \rrbracket] \rightarrow_h L' \llbracket v[p_1 \setminus u_1][p_2 \setminus u_2] \rrbracket$ and derivation $\Phi'_{L'}$ for the leftmost premise by the i.h. one gets $\Phi_{L'} \triangleright \Gamma_{L'} \vdash^{(b_l, e_l, m_l, f_l)} v[\langle p_1, p_2 \rangle \setminus L' \llbracket \langle u_1, u_2 \rangle \rrbracket] : \sigma$ where $b_l = b'_l$, $e_l = e'_l$, $m_l = m'_l + 1$ and $f_l = f'_l$. Furthermore $\Phi_{L'}$ is necessarily of the form:

$$\frac{\Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma \quad \Gamma_v|_p \Vdash^{(e_p, m_p, f_p)} p : [\times(\mathcal{A}_1, \mathcal{A}_2)] \quad \frac{\Gamma_u \vdash^{(b'_u, e'_u, m'_u, f'_u)} L' \llbracket u \rrbracket : \times(\mathcal{A}_1, \mathcal{A}_2)}{\Gamma_u \vdash^{(b'_u, e'_u, m'_u, f'_u)} L' \llbracket u \rrbracket : [\times(\mathcal{A}_1, \mathcal{A}_2)]}}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \vdash^{(b_v + b'_u, e_v + e'_u + e_p, m_v + m'_u + m_p, f_v + f'_u + f_p)} v[p \setminus L' \llbracket u \rrbracket] : \sigma}$$

where $b_l = b_v + b'_u$, $e_l = e_v + e'_u + e_p$, $m_l = m_v + m'_u + m_p$, $f_l = f_v + f'_u + f_p$ and $\Gamma_{L'} = \Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u$. Note that, by relevance and α -conversion we have that $\Gamma_{L'}|_q = \Gamma_u|_q$. Then one can construct the following derivation Φ_u :

$$\frac{\Gamma_u \vdash^{(b'_u, e'_u, m'_u, f'_u)} L' \llbracket u \rrbracket : \times(\mathcal{A}_1, \mathcal{A}_2) \quad \Gamma_u|_q \Vdash^{(e_q, m_q, f_q)} q : \mathcal{A} \quad \Phi_s}{\frac{\Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash^{(b'_u + b_s, e'_u + e_s + e_q, m'_u + m_s + m_q, f'_u + f_s + f_q)} L' \llbracket q \setminus s \rrbracket \llbracket u \rrbracket : \times(\mathcal{A}_1, \mathcal{A}_2)}{\Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash^{(b'_u + b_s, e'_u + e_s + e_q, m'_u + m_s + m_q, f'_u + f_s + f_q)} L' \llbracket q \setminus s \rrbracket \llbracket u \rrbracket : [\times(\mathcal{A}_1, \mathcal{A}_2)]}}$$

where $b_u = b'_u + b_s$, $e_u = e'_u + e_s + e_q$, $m_u = m'_u + m_s + m_q$ and $f_u = f'_u + f_s + f_q$. From $\Phi_v \triangleright \Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma$ and $\Pi_p \triangleright \Gamma_v|_p \Vdash^{(e_p, m_p, f_p)} p : [\times(\mathcal{A}_1, \mathcal{A}_2)]$ we build Φ :

$$\frac{\Phi_v \quad \Pi_p \quad \Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash^{(b_u, e_u, m_u, f_u)} L' \llbracket q \setminus s \rrbracket \llbracket u \rrbracket : [\times(\mathcal{A}_1, \mathcal{A}_2)]}{\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s \vdash^{(b_v + b_u, e_v + e_u + e_p, m_v + m_u + m_p, f_v + f_u + f_p)} v[p \setminus L' \llbracket q \setminus s \rrbracket \llbracket u \rrbracket] : \sigma}$$

With $\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u \parallel \mathbf{var}(q) \wedge \Gamma_s = (\Gamma_v \parallel \mathbf{var}(p) \wedge \Gamma_u) \parallel \mathbf{var}(q) \wedge \Gamma_s = \Gamma$. Furthermore,

$$* \quad b = b_v + b_u = b_v + b'_u + b_s = b_l + b_s = b'$$

$$* \quad e = e_v + e_u + e_p = e_v + e'_u + e_s + e_q + e_p = e_l + e_s + e_q = e'$$

$$* \quad m = m_v + m_u + m_p = m_v + m'_u + m_s + m_q + m_p = m_l + m_s + m_q = m' + 1$$

$$* \quad f = f_v + f_u + f_p = f_v + f'_u + f_s + f_q + f_p = f_l + f_s + f_q = f'$$

- Most of the inductive cases are straightforward, so we only show the interesting one. Let $t = v[p \setminus u] \rightarrow_h v[p \setminus u'] = t'$, where $v \not\rightarrow_h$ and $p \neq x$ and $u \rightarrow_h u'$. By construction there are subderivations $\Phi_v \triangleright \Gamma_v \vdash^{(b_v, e_v, m_v, f_v)} v : \sigma$, $\Gamma_v|_p \Vdash^{(e_p, m_p, f_p)} p : \mathcal{A}$ and $\Phi_{u'} \triangleright \Gamma_{u'} \vdash^{(b_{u'}, e_{u'}, m_{u'}, f_{u'})} u' : \mathcal{A}$ for some multiset \mathcal{A} and $\Gamma = (\Gamma_v \parallel \mathbf{var}(p)) \wedge \Gamma_{u'}$. Since p is not a variable then Φ_p ends with rule (pat_p) or (pat_\times) . In both cases \mathcal{A} contains only one type, let us say $\mathcal{A} = [\sigma_{u'}]$. Then $\Phi_{u'}$ has the following form

$$\Phi_{u'} \triangleright \frac{\Gamma_{u'} \vdash^{(b_{u'}, e_{u'}, m_{u'}, f_{u'})} u' : \sigma_{u'}}{\Gamma_{u'} \vdash^{(b_{u'}, e_{u'}, m_{u'}, f_{u'})} u' : [\sigma_{u'}]}$$

The i.h. applied to the premise of $\Phi_{u'}$ gives a derivation $\Gamma_{u'} \vdash^{(b_u, e_u, m_u, f_u)} u : \sigma_{u'}$ and having the expected counters. To conclude we build a type derivation Φ' for $v[p \setminus u']$ having the expected counters. \blacktriangleleft

Big Step Normalisation for Type Theory

Thorsten Altenkirch

School for Computer Science, University of Nottingham, UK
txa@cs.nott.ac.uk

Colin Geniet 

Computer Science Department, ENS Paris-Saclay, France
colin.geniet@ens-paris-saclay.fr

Abstract

Big step normalisation is a normalisation method for typed lambda-calculi which relies on a purely syntactic recursive evaluator. Termination of that evaluator is proven using a predicate called strong computability, similar to the techniques used to prove strong normalisation of β -reduction for typed lambda-calculi. We generalise big step normalisation to a minimalist dependent type theory. Compared to previous presentations of big step normalisation for e.g. the simply-typed lambda-calculus, we use a quotiented syntax of type theory, which crucially reduces the syntactic complexity introduced by dependent types. Most of the proof has been formalised using Agda.

2012 ACM Subject Classification Theory of computation \rightarrow Type theory

Keywords and phrases Normalisation, big step normalisation, type theory, dependent types, Agda

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.4

Supplementary Material <https://github.com/colingeniet/big-step-normalisation>

Funding *Thorsten Altenkirch*: supported by COST Action EUTypes CA15123 and USAF, Airforce office for scientific research, award FA9550-16-1-0029.

1 Introduction

1.1 Normalisation

In the context of typed lambda-calculi, normalisation refers to the process of computing a canonical representative, called normal form, in each $\beta\eta$ -equivalence class of terms. A very general definition of normalisation, previously used in e.g. [6, 3, 7], is the following. Normalisation is given by a set of normal forms and two (computable) maps: `norm` from terms to normal forms, and an embedding $\ulcorner _ \urcorner$ of normal forms into terms, satisfying

soundness If $u \simeq_{\beta\eta} v$, then $\text{norm } u = \text{norm } v$

completeness¹ For every term u , $\ulcorner \text{norm } u \urcorner \simeq_{\beta\eta} u$

stability For every normal form n , $\text{norm } \ulcorner n \urcorner = n$

The traditional way to define a normalisation function is through rewriting theory. One proves that $\beta\eta$ -reduction is confluent, and terminates² on typed terms. Normal forms are defined as terms which can not be $\beta\eta$ -reduced, and normalisation is done by reducing a term until reaching a normal form. Termination and confluence ensure the correctness of the definition. Soundness also follows from confluence, while completeness and stability are immediate. See for instance [16] for a detailed proof of this result for the simply-typed lambda-calculus and some variants (System F, System T). Unfortunately, problems arise for

¹ The choice of the words *soundness* and *completeness* comes from viewing normal forms as a model.

² We do not use the words strong or weak normalisation to refer to termination of the rewriting process, so as to avoid ambiguity with the generalised notion of normalisation introduced.



© Thorsten Altenkirch and Colin Geniet;

licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 4; pp. 4:1–4:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

some other variants of the lambda-calculus. For instance, the lambda-calculus with explicit substitutions does not terminate in the strong sense [22], and the lambda-calculus with coproduct types (i.e. disjoint union) is not confluent [14]. While some of these issues can be worked around, for instance by using weak termination and more restrictive reductions, these problems have led to the development of other methods.

One of them is *normalisation by evaluation* (NBE), introduced by Berger and Schwichtenberg [9] for the simply-typed lambda-calculus. The idea is to evaluate terms into a semantic model, meaning for instance that λ -abstractions (syntactic functions) are interpreted by actual (semantic) functions. A map from the model into normal forms is then defined, giving rise to the normalisation function by composition with evaluation. This method was for instance used to prove decidability of equivalence for the lambda-calculus with coproducts [4].

1.2 Big Step Normalisation

Big step normalisation (BSN) is a purely syntactic normalisation method, proposed in [3] by Chapman and the first author for the simply-typed lambda-calculus. The normalisation algorithm is in two parts. First, terms are evaluated by an environment machine, yielding syntactic values. Then, values are mapped to normal forms by a function named `quote`. Normalisation `norm` is done by evaluating in the identity environment, then applying `quote` on the resulting value. The embedding $\ulcorner _ \urcorner$ is the inclusion of normal forms into terms.

Evaluation and `quote` both have fairly simple definitions, but are not structurally recursive, hence their termination is not obvious. To prove termination, a Tait-style predicate [25] called *strong computability* (SC) is defined on values:

- A value v of the base type is strongly computable if normalisation terminates on v .
 - A value f of a function type is SC if it preserves SC when applied to an argument.
- The following results can then be proved.
- `quote` terminates on any SC value, and conversely any neutral value (i.e. a value which is not a λ -abstraction) on which `quote` terminates is SC.
 - In a SC environment, evaluation terminates and yields SC results.

Termination of `norm` follows from these results. Completeness and stability are straightforward. The proof of soundness is more involved, and shares some similarities with the proof of termination, but replaces strong computability with a binary relation on values.

1.3 BSN for Type Theory and Quotiented Syntax

Chapman also considered BSN for dependent type theory in [10], but did not provide a full proof of correctness, due to the syntactic complexity added by dependent types.

In this work, we propose some methods to simplify the proof of BSN in the case of dependent types, allowing us to complete it. Notably, we use the quotiented syntax of type theory proposed in [8]. By only considering terms quotiented by $\beta\eta$ -equivalence, the syntax becomes significantly lighter. For instance, the coercion constructors which form a large part of the syntactic boilerplate encountered in [10] become unnecessary.

With a quotiented syntax, the notion of normalisation changes slightly. If $\simeq_{\beta\eta}$ is replaced with equality of quotiented terms in the first definition of a normalisation function, then soundness simply states that `norm` is correctly defined on the quotiented syntax, while completeness and stability state that `norm` and $\ulcorner _ \urcorner$ are inverse of each other. This leads to the following definition proposed in [7]: a normalisation function is simply an isomorphism between quotiented terms and normal forms. Obviously, this definition requires a sensible

notion of normal forms – one can not consider quotiented terms to be normal forms, and identity to be normalisation. Thus, we require normal forms to have a simple inductive definition, which ensures decidability of equality.³

1.4 Structure of the Paper

Section 2 presents the metatheory, notation, and conventions used in this paper. Section 3 presents the quotiented syntax of type theory. Section 4 introduces a notion of weakening of contexts. Section 5 defines big step normalisation itself. Because it is not a priori clear that BSN defines a correct function (termination for instance is problematic) we formally define normalisation by its big step semantics, i.e. as a relation between inputs and output. Section 6 focuses on the two major correctness proofs: termination and soundness. The proof of termination remains similar to the case of simple types. The main difference is that we develop a simplified and generalised induction principle for types, which allows us to manipulate dependent types in almost the same way as simple types during the proof. The proof of soundness for an unquotiented syntax seems much harder to adapt, we instead provide a simple proof using soundness of NBE. Finally, Section 7 explains how the proof of BSN can be adapted to a cubical metatheory, using higher inductive types to encode quotient inductive types.

1.5 Related Work

Big step semantics have previously been used for the purpose of normalisation. For instance T. Coquand uses a big step relation to decide conversion in type theory [12], but relies on considerations on untyped terms, and focuses on deciding conversion, rather than fully normalising terms. P.B. Levy uses Tait’s method to prove termination of a big step semantics in the case of a simple programming language [21]. Big step normalisation was developed by Chapman and the first author for a combinatory calculus [2], and for the simply-typed lambda-calculus [3]. A generalisation to type theory was proposed [10], but without a full proof of correctness. The present paper can be seen as a continuation of these works.

An important difference compared to previous works on big step normalisation is that we use a quotiented syntax of type theory. This builds upon the work by Kaposi and the first author which provides a concise, quotiented syntax of type theory within (a larger) type theory [8], and formalises normalisation by evaluation in this syntax [7]. This quotiented syntax is closely related to categories with families [15, 18], in that the syntax is essentially an initial category with families. The syntax is formalised using quotient inductive-inductive types (QIIT), which were previously used in [24] – although not under that name – to e.g. define Cauchy reals in type theory. More recently, the precise notion and semantics of QIIT has been the subject of work such as [1, 13, 20].

2 Metatheory and Notations

The present work has been formalised using a cubical metatheory [11] implemented by Agda [23]. This cubical theory provides a simple way to define quotient inductive inductive types (QIIT, cf. [8]) as a special case of higher inductive types. However, for simplicity, we

³ In the unquotiented case, the embedding of normal forms into terms (which can be proved to be injective) ensures that equality of normal forms is decidable, hence why no such restriction was required.

4:4 Big Step Normalisation for Type Theory

prefer to present this paper in a strict, intentional Martin-Löf Type Theory, extended with QIIT. Functional extensionality is assumed, and can in fact be proved using the interval quotient type. See Section 7 for a discussion of the implementation in a cubical metatheory.

Our metatheoretic notations are loosely based on the syntax of Agda. Function types are written as $(x : A) \rightarrow B$, or simply $A \rightarrow B$ for non-dependent functions. We use infix arguments denoted by underscores, e.g. $_ , _$ applied to x and y is written as x, y . Functions with implicit arguments are defined as $f : \{x : A\} \rightarrow B$, and the argument can be either omitted, or given in subscript as f_x . Sum types (dependent pairs) are denoted by $\Sigma(x : A), B$. We denote by **Set** the universe of types, and by **Prop** the universe of mere propositions, i.e. the types in which all elements are equal. The equality type is denoted by $x \equiv y$, while $=$ is only used in definitions. The transport of $x : P a$ along an equality $p : a \equiv b$ is denoted by $p_* x : P b$. If $p : a \equiv b$, the type of dependent equalities between $x : P a$ and $y : P b$ lying over p is denoted by $x \equiv^P y$. For simplicity and readability, transports and dependent equality types will be omitted starting from Section 4.

Inductive types are introduced by **data**, the sort of the defined type, and the signatures of the constructors. Inductive functions are defined by pattern-matching. For instance:

data $\mathbb{N} : \mathbf{Set}$ where	$_ + _ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$
$0 : \mathbb{N}$	$n + 0 = n$
$S : \mathbb{N} \rightarrow \mathbb{N}$	$n + (S m) = S (n + m)$

We allow a very general form of mutual induction, called *inductive-inductive* definitions. A good example is the following fragment of the syntax of dependent types from next section.

data $\mathbf{Con} : \mathbf{Set}$ where	data $\mathbf{T}_y : \mathbf{Con} \rightarrow \mathbf{Set}$ where
$\bullet : \mathbf{Con}$	$\mathbf{U} : \mathbf{T}_y \Gamma$
$_, _ : (\Gamma : \mathbf{Con}) \rightarrow \mathbf{T}_y \Gamma \rightarrow \mathbf{Con}$	$\mathbf{II} : (A : \mathbf{T}_y \Gamma) \rightarrow \mathbf{T}_y (\Gamma, A) \rightarrow \mathbf{T}_y \Gamma$

In addition to \mathbf{Con} and \mathbf{T}_y being defined simultaneously, note that \mathbf{T}_y is a family indexed by \mathbf{Con} , and the signature of the constructor \mathbf{II} from \mathbf{T}_y uses the constructor $_, _$ from \mathbf{Con} .

QIIT furthermore allow *equality* or *quotient* constructors, which build of equalities in the defined type. For example the interval type is defined by two endpoints and an equality:

data $\mathbb{I} : \mathbf{Set}$ **where**

$a : \mathbb{I}$

$b : \mathbb{I}$

$p : a \equiv b$

A function defined by induction on a QIIT must be defined inductively on regular constructors, and must respect all quotient constructors, meaning that it must map the elements equated by a quotient constructors to images which are provably equal. For instance, to define a function by induction on \mathbb{I} , one must specify the images $f(a)$ and $f(b)$, then prove that $f(a) \equiv f(b)$. The reader may refer to [8] for more details on QIIT.

Finally, all free variables in definitions and lemmas are implicitly universally quantified. Omitted types can be inferred from the context and the naming conventions.

3 Quotiented Syntax of Type Theory

This section introduces the syntax of type theory based on QIIT proposed by Kaposi and the first author in [8, 7]. The reader should refer to the former for further details.

This syntax is intrinsically typed, with De Bruijn indices, and explicit substitutions. Contexts, types, substitutions and terms are mutually defined. We denote contexts by $\Gamma, \Delta, \Theta, \Phi$, types by A, B, C , substitutions by σ, ν, δ , and terms by s, t, u .

```

data Con : Set                               Con is the set of contexts
data Ty  : Con → Set                          Ty  $\Gamma$  are the types in context  $\Gamma$ 
data Sub : Con → Con → Set                   Sub  $\Gamma \Delta$  are the substitutions from  $\Delta$  to  $\Gamma$ 
data Tm  : ( $\Gamma$  : Con) → Ty  $\Gamma$  → Set     Tm  $\Gamma A$  are the terms of type  $A$  in  $\Gamma$ 

```

Syntax constructors follow closely the definition of a category with families [15, 18] with product types. Contexts and substitutions form a category, types are a presheaf, and terms are a family of presheaves over types. The constructor for dependent function types is denoted by Π . There is a base type \mathbf{U} , and a base dependent family \mathbf{El} indexed by \mathbf{U} . One may see \mathbf{U} as a universe, i.e. a type whose elements are types, when interpreted through \mathbf{El} – this is reflected by the names of the constructors. Because we consider a minimalist type theory, it is only an abstract universe, meaning that no element of \mathbf{U} can be built in a closed context. However, one may use contexts to postulate the existence of types in \mathbf{U} .

The syntax constructors are listed below, with regular constructors on the left, and equality constructors on the right.

data Con where

- \bullet : Con
- $_ , _$: (Γ : Con) → Ty Γ → Con

data Ty where

```

 $\_$ [ $\_$ ] : Ty  $\Delta$  → Sub  $\Gamma \Delta$  → Ty  $\Gamma$ 
 $\mathbf{U}$     : Ty  $\Gamma$ 
 $\mathbf{El}$    : Tm  $\Gamma \mathbf{U}$  → Ty  $\Gamma$ 
 $\Pi$     : ( $A$  : Ty  $\Gamma$ ) → Ty ( $\Gamma, A$ ) → Ty  $\Gamma$ 

```

data Sub where

```

id      : Sub  $\Gamma \Gamma$ 
 $\_ \circ \_$  : Sub  $\Delta \Theta$  → Sub  $\Gamma \Delta$  → Sub  $\Gamma \Theta$ 
 $\epsilon$    : Sub  $\Gamma \bullet$ 
 $\_ , \_$   : ( $\sigma$  : Sub  $\Gamma \Delta$ ) → Tm  $\Gamma A$ [ $\sigma$ ]
        → Sub  $\Gamma (\Delta, A)$ 
 $\pi_1$    : Sub  $\Gamma (\Delta, A)$  → Sub  $\Gamma \Delta$ 

```

data Tm where

```

 $\pi_2$    : ( $\sigma$  : Sub  $\Gamma (\Delta, A)$ ) → Tm  $\Gamma (A[\pi_1 \sigma])$ 
 $\_$ [ $\_$ ]   : Tm  $\Delta A$  → ( $\sigma$  : Sub  $\Gamma \Delta$ )
        → Tm  $\Gamma A$ [ $\sigma$ ]
 $\lambda$    : Tm ( $\Gamma, A$ )  $B$  → Tm  $\Gamma (\Pi A B)$ 
app     : Tm  $\Gamma (\Pi A B)$  → Tm ( $\Gamma, A$ )  $B$ 

```

data Ty where

```

[id] : A[id]  $\equiv$  A
[ $\circ$ ] : A[ $\sigma \circ \nu$ ]  $\equiv$  A[ $\sigma$ ][ $\nu$ ]
 $\mathbf{U}[\_]$  :  $\mathbf{U}[\sigma] \equiv \mathbf{U}$ 
 $\mathbf{El}[\_]$  : ( $\mathbf{El} u$ )[ $\sigma$ ]  $\equiv$   $\mathbf{El}(\mathbf{U}[\_]*u[\sigma])$ 
 $\Pi[\_]$  : ( $\Pi A B$ )[ $\sigma$ ]  $\equiv$   $\Pi(A[\sigma])(B[\sigma \uparrow A])$ 

```

data Sub where

```

id $\circ$  : id  $\circ$   $\sigma \equiv \sigma$ 
 $\circ$  id :  $\sigma \circ$  id  $\equiv \sigma$ 
 $\circ \circ$  : ( $\sigma \circ \nu$ )  $\circ$   $\delta \equiv \sigma \circ (\nu \circ \delta)$ 
 $\epsilon \eta$  : { $\sigma$  : Sub  $\Gamma \bullet$ } →  $\sigma \equiv \epsilon$ 
 $\pi_1 \beta$  :  $\pi_1(\sigma, u) \equiv \sigma$ 
 $\pi_1 \eta$  :  $\pi_1 \sigma, \pi_2 \sigma \equiv \sigma$ 
 $\circ , \circ$  : ( $\sigma, u$ )  $\circ \nu \equiv$  ( $\sigma \circ \nu$ ), ( $[\circ]^{-1} u[\nu]$ )

```

data Tm where

```

 $\pi_2 \beta$  :  $\pi_2(\sigma, u) \equiv \pi_1^\beta u$ 
 $\beta$       : app ( $\lambda u$ )  $\equiv u$ 
 $\eta$       :  $\lambda$ (app  $u$ )  $\equiv u$ 
 $\lambda[\_]$    : ( $\lambda u$ )[ $\sigma$ ]  $\equiv$   $\Pi[\_]$   $\lambda(u[\sigma \uparrow A])$ 

```

4:6 Big Step Normalisation for Type Theory

Equations $\Pi[]$ and $\lambda[]$ use the lifting of a substitution by a type, defined as follows.

$$\begin{aligned} _ \uparrow _ &: (\sigma : \text{Sub } \Gamma \Delta) \rightarrow (A : \text{Ty } \Delta) \rightarrow \text{Sub } (\Gamma, A[\sigma]) (\Delta, A) \\ \sigma \uparrow A &= (\sigma \circ \pi_1 \text{ id}), ([\sigma]^{-1} \ast \pi_2 \text{ id}) \end{aligned}$$

This syntax uses a categorical application constructor `app`, which is essentially the inverse of λ . One may understand `app f` as the application of f to a fresh variable. In order to obtain the usual application, denoted $_ \$ _$, this fresh variable must be substituted by the argument. We denote this substitution of the last variable in the context by $\langle _ \rangle$.

$$\begin{aligned} \langle _ \rangle &: \text{Tm } \Gamma A \rightarrow \text{Sub } \Gamma (\Gamma, A) \\ \langle u \rangle &= \text{id}, [\text{id}]^{-1} \ast u \\ _ \$ _ &: \text{Tm } \Gamma (\Pi A B) \rightarrow (u : \text{Tm } \Gamma A) \rightarrow \text{Tm } \Gamma (B[\langle u \rangle]) \\ f \$ u &= (\text{app } f)[\langle u \rangle] \end{aligned}$$

As a simple example, let us translate the lambda term $\lambda x^A. \lambda y^B. x$ to this syntax. We assume that A is type in context Γ , and B a type in context (Γ, A) (or in short $\Gamma : \text{Con}$, $A : \text{Ty } \Gamma$, $B : \text{Ty } (\Gamma, A)$), so that (Γ, A, B) is a context. We start with `id`, intuitively the substitution containing all variables in context.

$$\text{id} : \text{Sub } (\Gamma, A, B) (\Gamma, A, B)$$

The second to last variable in the context, corresponding to x , is retrieved through projections.

$$\pi_2(\pi_1 \text{ id}) : \text{Tm } (\Gamma, A, B) A[\pi_1 \text{ id}]$$

Finally, the lambda-abstractions are added.

$$\lambda(\lambda(\pi_2(\pi_1 \text{ id}))) : \text{Tm } \Gamma (\Pi A (\Pi B A[\pi_1 \text{ id}])))$$

4 Weakenings

In this section, we introduce variables and weakenings of contexts. The presentation is the same as in [7], except that the latter uses the name “renamings” instead.

Variables, denoted by x, y, z , are defined as typed De Bruijn indices, with constructors `vz` and `vs` standing for “0” and successor. Variables can be embedded into terms by applying projections to `id` – intuitively, `id` is the substitution formed by all the variables in context.

$$\begin{aligned} \text{data Var} &: (\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Set} \text{ where} & \ulcorner _ \urcorner &: \text{Var } \Gamma A \rightarrow \text{Tm } \Gamma A \\ \text{vz} &: \text{Var } (\Gamma, A) (A[\pi_1 \text{ id}]) & \ulcorner \text{vz} \urcorner &= \pi_2 \text{ id} \\ \text{vs} &: \text{Var } \Gamma A \rightarrow \text{Var } (\Gamma, B) (A[\pi_1 \text{ id}]) & \ulcorner \text{vs } x \urcorner &= \ulcorner x \urcorner[\pi_1 \text{ id}] \end{aligned}$$

Weakening substitutions (or simply weakenings), denoted by α, β, γ , are substitutions composed only of variables. This regroups the usual notions of weakening (i.e. forgetting a variable), contraction, and reordering of independent variables. Note that constructors ϵ and $_ , _$ are overloaded due to the similarity with substitutions.

$$\begin{aligned} \text{data Wk} &: \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \text{ where} & \ulcorner _ \urcorner &: \text{Wk } \Gamma \Delta \rightarrow \text{Sub } \Gamma \Delta \\ \epsilon &: \text{Wk } \Gamma \bullet & \ulcorner \epsilon \urcorner &= \epsilon \\ _ , _ &: (\alpha : \text{Wk } \Gamma \Delta) \rightarrow \text{Var } \Gamma A[\ulcorner \alpha \urcorner] \rightarrow \text{Wk } \Gamma (\Delta, A) & \ulcorner \alpha, x \urcorner &= \ulcorner \alpha \urcorner, \ulcorner x \urcorner \end{aligned}$$

Unlike regular substitutions, identity and composition of weakenings are not constructors, but inductive definitions. Some auxiliary functions are required: wk weakens the context of a weakening substitution by a type A , and $\llbracket _ \rrbracket$ applies a weakening substitution to a variable. These functions all commute with embeddings of variables and weakenings. We omit the inductive definitions and proofs, which are simple.

$$\begin{array}{ll}
\text{wk} & : (A : \text{Ty } \Gamma) \rightarrow \text{Wk } \Gamma \Delta \rightarrow \text{Wk } (\Gamma, A) \Delta & \ulcorner \text{wk} \urcorner : \ulcorner \text{wk } A \alpha \urcorner \equiv \ulcorner \alpha \urcorner \circ (\pi_1 \text{id}) \\
\text{id} & : \{\Gamma : \text{Con}\} \rightarrow \text{Wk } \Gamma \Gamma & \ulcorner \text{id} \urcorner : \ulcorner \text{id} \urcorner \equiv \text{id} \\
\llbracket _ \rrbracket & : \text{Var } \Delta A \rightarrow (\alpha : \text{Wk } \Gamma \Delta) \rightarrow \text{Var } \Gamma (A[\ulcorner \alpha \urcorner]) & \ulcorner \llbracket _ \rrbracket \urcorner : \ulcorner x[\alpha] \urcorner \equiv \ulcorner x \urcorner [\ulcorner \alpha \urcorner] \\
\llbracket _ \circ _ \rrbracket & : \text{Wk } \Delta \Theta \rightarrow \text{Wk } \Gamma \Delta \rightarrow \text{Wk } \Gamma \Theta & \ulcorner \circ \urcorner : \ulcorner \alpha \circ \beta \urcorner \equiv \ulcorner \alpha \urcorner \circ \ulcorner \beta \urcorner
\end{array}$$

Contexts and weakenings form a category with these operations. Types, terms, and substitutions can be weakened by applying a weakening substitution, seen as a regular substitution through embedding. These operations respects identity and composition, that is types and substitutions are presheaves on the category of weakenings, while terms are a family of presheaves over types. Definitions are below, with the lemmas on the right (proofs omitted).

$$\begin{array}{ll}
\llbracket _ \rrbracket^{+-} & : \text{Ty } \Delta \rightarrow \text{Wk } \Gamma \Delta \rightarrow \text{Ty } \Gamma & +\text{id} : A^{+\text{id}} \equiv A \\
A^{+\alpha} & = A[\ulcorner \alpha \urcorner] & +\circ : A^{+(\alpha \circ \beta)} \equiv (A^{+\alpha})^{+\beta} \\
\llbracket _ \rrbracket^{+-} & : \text{Tm } \Delta A \rightarrow (\alpha : \text{Wk } \Gamma \Delta) \rightarrow \text{Tm } \Gamma A^{+\alpha} & +\text{id} : u^{+\text{id}} \equiv u \\
u^{+\alpha} & = u[\ulcorner \alpha \urcorner] & +\circ : u^{+(\alpha \circ \beta)} \equiv (u^{+\alpha})^{+\beta} \\
\llbracket _ \rrbracket^{+-} & : \text{Sub } \Delta \Theta \rightarrow \text{Wk } \Gamma \Delta \rightarrow \text{Sub } \Gamma \Theta & +\text{id} : \sigma^{+\text{id}} \equiv \sigma \\
\sigma^{+\alpha} & = \sigma \circ \ulcorner \alpha \urcorner & +\circ : \sigma^{+(\alpha \circ \beta)} \equiv (\sigma^{+\alpha})^{+\beta}
\end{array}$$

This will be a general pattern in later constructions and proofs: families of sets (e.g. values, normal forms, ...) have a presheaf-like structure, which simply means that the elements can be weakened coherently. Similarly, functions are natural transformations, i.e. commute with weakening, and predicates are sub-presheaves, i.e. are stable under weakening. The corresponding definitions and proofs are typically straightforward, and we will often not mention them. We abusively denote all applications of weakenings by $\llbracket _ \rrbracket^{+-}$.

Finally, given a type A , one may consider $\text{wk } A \text{id} : \text{Wk } (\Gamma, A) \Gamma$, the weakening of the context Γ by A . We abuse notations and write u^{+A} for $u^{+(\text{wk } A \text{id})}$.

5 Normalisation Relation

This section defines the big step normalisation algorithm using the previous syntax of type theory. As further explained in Section 5.2, this algorithm can not yet be formally defined as a function. Thus, it is defined as a relation in order to carry out the correctness proof.

We first define values and the evaluation from terms to values, then normal forms and the function quote mapping values to normal forms. Normalisation is done by applying evaluation followed by quote.

5.1 Values

A value is either a closure, corresponding to the delayed evaluation of a lambda-abstraction, or a neutral value, that is the stuck application of a variable to values. We define mutually values (denoted by v, w), neutral values (denoted by n), and environments (substitutions composed of values, denoted by ρ, ω), together with the associated embeddings.

4:8 Big Step Normalisation for Type Theory

<p>data Val : (Γ : Con) → Ty Γ → Set where</p> <p> neu : NV Γ A → Val Γ A</p> <p> clos : Tm (Δ, A) B → (ρ : Env Γ Δ)</p> <p> → Val Γ ((Π A B)[$\ulcorner \rho \urcorner$])</p> <p>data NV : (Γ : Con) → Ty Γ → Set where</p> <p> var : Var Γ A → NV Γ A</p> <p> app : NV Γ (Π A B) → (v : Val Γ A)</p> <p> → NV Γ (B[$\langle \ulcorner v \urcorner \rangle$])</p> <p>data Env : Con → Con → Set where</p> <p> ε_• : Env Γ •</p> <p> _,_ : (ρ : Env Γ Δ) → Val Γ (A[$\ulcorner \rho \urcorner$]) → Env Γ (Δ, A)</p>	<p>$\ulcorner _ \urcorner$: Val Γ A → Tm Γ A</p> <p>$\ulcorner \text{neu } n \urcorner = \ulcorner n \urcorner$</p> <p>$\ulcorner \text{clos } u \rho \urcorner = (\lambda u)[\ulcorner \rho \urcorner]$</p> <p>$\ulcorner _ \urcorner$: NV Γ A → Tm Γ A</p> <p>$\ulcorner \text{var } x \urcorner = \ulcorner x \urcorner$</p> <p>$\ulcorner \text{app } n v \urcorner = \ulcorner n \urcorner \\$ \ulcorner v \urcorner$</p> <p>$\ulcorner _ \urcorner$: Env Γ Δ → Sub Γ Δ</p> <p>$\ulcorner \epsilon \urcorner = \epsilon$</p> <p>$\ulcorner \rho, v \urcorner = \ulcorner \rho \urcorner, \ulcorner v \urcorner$</p>
--	---

This definition has an issue when used with a quotiented syntax: values can be equivalent as terms (formally, have equal embeddings), but not equal. For instance, in a closure $\text{clos } u \rho$, if the body u never refers to the environment ρ , then modifying ρ yields a distinct but equivalent value. Then, evaluation would map equivalent terms to distinct values, hence could not be defined on the quotiented syntax. This is fixed by forcing equivalent values to be equal with the following quotient constructor.

data Val **where**

 qVal : (v w : Val Γ A) → $\ulcorner v \urcorner \equiv \ulcorner w \urcorner \rightarrow v \equiv w$

The corresponding result for environments can be proved by induction on contexts.

qEnv : (ρ ω : Env Γ Δ) → $\ulcorner \rho \urcorner \equiv \ulcorner \omega \urcorner \rightarrow \rho \equiv \omega$

Weakening is defined by induction on values, neutral values, and environments, we omit the definitions and the associated lemmas. Finally, the identity environment is defined by induction on the context, and uses weakening of environments.

idenv : {Γ : Con} → Env Γ Γ

idenv_• = ε

idenv_{Γ,A} = idenv_Γ^{+A}, neu (var vz)

5.2 Evaluation

The first stage of normalisation is an environment machine, which evaluate terms in an environment, and returns values. It consists of three mutually defined functions: **eval** and **evals** evaluate terms and substitutions respectively in an environment, while **__@__** computes the application of a value to another.

eval : Tm Δ A → (ρ : Env Γ Δ) → Val Γ A[$\ulcorner \rho \urcorner$]

eval (π₂ σ) ρ = **let** (ω, v) = (evals σ ρ) **in** v

eval (u[σ]) ρ = eval u (evals σ ρ)

eval (λu) ρ = clos u ρ

eval (app u) (ρ, v) = (eval u ρ) @ v

$$\begin{aligned}
& \text{evals} : \text{Sub } \Delta \Theta \rightarrow \text{Env } \Gamma \Delta \rightarrow \text{Env } \Gamma \Theta \\
& \text{evals id } \rho = \rho \\
& \text{evals } (\sigma \circ \nu) \rho = \text{evals } \sigma (\text{evals } \nu \rho) \\
& \text{evals } \epsilon \rho = \epsilon \\
& \text{evals } (\sigma, u) \rho = (\text{evals } \sigma \rho), (\text{eval } u \rho) \\
& \text{evals } (\pi_1 \sigma) \rho = \mathbf{let} (\omega, v) = (\text{evals } \sigma \rho) \mathbf{in} \omega \\
& _@_ : \text{Val } \Gamma (\Pi A B) \rightarrow (v : \text{Val } \Gamma A) \rightarrow \text{Val } \Gamma B[\langle \ulcorner v \urcorner \rangle] \\
& (\text{clos } u \rho) @ v = \text{eval } u (\rho, v) \\
& (\text{neu } n) @ v = \text{neu } (\text{app } n v)
\end{aligned}$$

Most cases are straightforward. Note how evaluation of a lambda simply returns a closure, delaying the evaluation of the body. The latter occurs in the first case of $_@_$, as the application of a closure to a value is computed by evaluating the body of the closure in the extended environment. Evaluation of the projections π_1, π_2 performs a projection on an environment, expressed through the $\mathbf{let} \dots \mathbf{in}$ construct with an obvious meaning.

However, there are several problems with this presentation of the evaluator. Firstly, the functions are defined by recursion on terms and substitutions, which are QIIT, but we did not bother to verify that equality constructors are respected. Perhaps more worryingly, the function is not structurally recursive: the last case of eval applies $_@_$ to $\text{eval } u \rho$, which a priori is an arbitrary value. Thus it is not clear that the evaluator terminates.

The proof of correctness of this algorithm is not trivial, and is the subject of Section 6. For now, we will only define the algorithm, i.e. we consider the previous definition as a *programming* function, rather than an (incorrect) mathematical function. In order to formally define this algorithm, we represent it by its big step semantics, that is the relation between inputs and outputs of the evaluator. For instance, we denote by $\text{eval } t \rho \Downarrow v$ the proposition “ t evaluates to v in environment ρ ”.

$$\begin{aligned}
& \mathbf{data} \text{eval_}_ \Downarrow _ : \text{Tm } \Delta A \rightarrow \text{Env } \Gamma \Delta \rightarrow \text{Val } \Gamma B \rightarrow \text{Prop} \mathbf{where} \\
& \quad \text{eval}\pi_2 : \text{evals } \sigma \rho \Downarrow (\omega, v) \rightarrow \text{eval } (\pi_2 \sigma) \rho \Downarrow v \\
& \quad \text{eval}[] : \text{evals } \sigma \rho \Downarrow \omega \rightarrow \text{eval } u \omega \Downarrow v \rightarrow \text{eval } (u[\sigma]) \rho \Downarrow v \\
& \quad \text{eval}\lambda : \text{eval } (\lambda u) \rho \Downarrow (\text{clos } u \rho) \\
& \quad \text{evalapp} : \text{eval } f \rho \Downarrow g \rightarrow g @ v \Downarrow w \rightarrow \text{eval } (\text{app } f) (\rho, v) \Downarrow w \\
& \mathbf{data} \text{evals_}_ \Downarrow _ : \text{Sub } \Delta \Theta \rightarrow \text{Env } \Gamma \Delta \rightarrow \text{Env } \Gamma \Theta \rightarrow \text{Prop} \mathbf{where} \\
& \quad \text{evalsid} : \text{evals id } \rho \Downarrow \rho \\
& \quad \text{evals}\circ : \text{evals } \nu \rho \Downarrow \omega \rightarrow \text{evals } \sigma \omega \Downarrow \xi \rightarrow \text{evals } (\sigma \circ \nu) \rho \Downarrow \xi \\
& \quad \text{evals}\epsilon : \text{evals } \epsilon \rho \Downarrow \epsilon \\
& \quad \text{evals}, : \text{evals } \sigma \rho \Downarrow \omega \rightarrow \text{eval } u \rho \Downarrow v \rightarrow \text{evals } (\sigma, u) \rho \Downarrow (\omega, v) \\
& \quad \text{evals}\pi_1 : \text{evals } \sigma \rho \Downarrow (\omega, v) \rightarrow \text{eval } (\pi_1 \sigma) \rho \Downarrow \omega \\
& \mathbf{data} _@_ \Downarrow _ : \text{Val } \Gamma A \rightarrow \text{Val } \Gamma B \rightarrow \text{Val } \Gamma C \rightarrow \text{Prop} \mathbf{where} \\
& \quad @\text{clos} : \text{eval } u (\rho, v) \Downarrow w \rightarrow (\text{clos } u \rho) @ v \Downarrow w \\
& \quad @\text{neu} : (\text{neu } n) @ v \Downarrow (\text{neu } (\text{app } n v))
\end{aligned}$$

The types of the above relations may seem surprisingly imprecise. For instance, the type of eval does not give any information on the type of the return value – it is a value of some unknown type B – whereas we know that it should have type $A[\langle \ulcorner \rho \urcorner \rangle]$ when evaluating

4:10 Big Step Normalisation for Type Theory

in environment ρ . Similarly, we do not even require the first argument of $@$ to be a function. It would be possible to define the evaluation relation with more restrictive types, but this would only complicate later proofs by requiring many additional transports. This choice may be compared to heterogeneous equality, which can similarly simplify proofs merely by being less restrictive than dependent equality types.

Of course, the expected type restrictions on evaluation can still be proved as lemmas.

► **Lemma 1.**

$$\frac{\text{eval } u \ \rho \Downarrow v}{\ulcorner v \urcorner \equiv u[\ulcorner \rho \urcorner]} \quad \frac{\text{evals } \sigma \ \rho \Downarrow \omega}{\ulcorner \omega \urcorner \equiv \sigma \circ \ulcorner \rho \urcorner} \quad \frac{f \ @ \ v \Downarrow w}{\ulcorner f \urcorner \$ \ulcorner v \urcorner \equiv \ulcorner w \urcorner}$$

Proof. By simultaneous induction on the definitions of the relations `eval`, `evals`, and `@`. ◀

A soundness property follows.

► **Lemma 2.**

$$\frac{\text{eval } u \ \rho \Downarrow v \quad \text{eval } u \ \rho \Downarrow w}{v \equiv w} \quad \frac{\text{evals } \sigma \ \rho \Downarrow \omega \quad \text{evals } \sigma \ \rho \Downarrow \delta}{\omega \equiv \delta}$$

$$\frac{f \ @ \ u \Downarrow v \quad f \ @ \ u \Downarrow w}{v \equiv w}$$

Proof. Using Lemma 1, and that embeddings of values and environments are injective by `qVal` and `qEnv`. ◀

5.3 Normal Forms

Having defined the evaluator, we continue with the function `quote` which maps values to normal forms. The classic notion of *η -long β -normal forms* (see for instance [19]) is used, which interestingly is shared with normalisation by evaluation (cf. [7]).

Like values, normal forms are defined mutually with neutral normal forms, i.e. the application of a variable to normal forms. An important difference is that not all neutral normal forms are normal forms: it is only true for neutral normal forms of the base types (i.e. `U` and `El`). This restriction ensures that normal forms are sufficiently η -expanded.

$$\begin{array}{ll} \text{data Nf} : (\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Set} \text{ where} & \ulcorner _ \urcorner : \text{Nf } \Gamma \ A \rightarrow \text{Tm } \Gamma \ A \\ \lambda \quad : \text{Nf } (\Gamma, A) \ B \rightarrow \text{Nf } \Gamma (\Pi \ A \ B) & \ulcorner \lambda n \urcorner \quad = \lambda \ulcorner n \urcorner \\ \text{neuU} : \text{NN } \Gamma \ \text{U} \rightarrow \text{Nf } \Gamma \ \text{U} & \ulcorner \text{neuU } n \urcorner = \ulcorner n \urcorner \\ \text{neuEl} : \text{NN } \Gamma \ (\text{El } u) \rightarrow \text{Nf } \Gamma \ (\text{El } u) & \ulcorner \text{neuEl } n \urcorner = \ulcorner n \urcorner \\ \text{data NN} : (\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Set} \text{ where} & \ulcorner _ \urcorner : \text{NN } \Gamma \ A \rightarrow \text{Tm } \Gamma \ A \\ \text{var} : \text{Var } \Gamma \ A \rightarrow \text{NN } \Gamma \ A & \ulcorner \text{var } x \urcorner \quad = \ulcorner x \urcorner \\ \text{app} : \text{NN } \Gamma \ (\Pi \ A \ B) \rightarrow (n : \text{NN } \Gamma \ A) & \ulcorner \text{app } m \ n \urcorner = \ulcorner m \urcorner \$ \ulcorner n \urcorner \\ & \rightarrow \text{NN } \Gamma \ (B[\ulcorner n \urcorner >]) \end{array}$$

Note that normal forms are indexed by regular types, we do not use a notion of normal types. Indeed, normalising types and terms simultaneously only seems to complicate matters, and it is easier to first normalise a term without worrying about its type, then recursively normalise the type. A disadvantage of this choice is that equality of normal forms is not a priori decidable, because it would require to test equality of types, and in turn equality of terms. This issue can be solved once the normalisation function is defined by proving decidability of equality for terms, normal forms, and types simultaneously, as shown in [7].

5.4 Quote

The function `quote` is defined by induction on the type of the value, together with `quoten` which maps neutral values to neutral normal forms by recursively applying `quote`. Like the evaluator, we begin with an informal definition as a function, which is then translated to a relation.

```

quote : {A : Ty} → Val Γ A → Nf Γ A
quoteU (neu v)    = neuU (quoten v)
quote(El t) (neu v) = neuEl (quoten v)
quote(Π A B) f     = λ(quote (f+A @ neu (var vz)))
quoten : NV Γ A → NN Γ A
quoten (var x)    = var x
quoten (app f v) = app (quoten f) (quote v)

```

A value of a base type is necessarily neutral, hence it suffice to use `quoten` in that case. For function types, the definition of normal forms requires the result to be an abstraction. This is done by η -expanding the value, and applying `quote` to the body of the resulting abstraction. The η -expansion is somewhat technical to define. First, the function is weakened as f^{+A} to allow the introduction of a new variable of type A represented by the De Bruijn index `vz`. This variable is turned into a value by the `var` and `neu` constructors, and the weakened function is applied using `@`, giving the body of the η -expansion.

Beside the problems of termination and correctness with regards to quotient constructors which already appeared in the evaluator, one may note that `quote` is not defined on the `_[_]` type constructor. We will later show that the definition for `_[_]` can in fact be inferred from the other cases and the equality constraints. For now we again ignore all issues by considering the big step semantics of `quote`.

```

data quote : Val Γ A → Nf Γ A → Prop where
  quoteU : {v : NV U} → quoten v ↓ n → quote (neu v) ↓ (neu n)
  quoteEl : {v : NV (El t)} → quoten v ↓ n → quote (neu v) ↓ (neu n)
  quoteΠ : f+A @ (neu (var vz)) ↓ v → quote v ↓ n → quote f ↓ (λ n)
data quoten : NV Γ A → NN Γ A → Prop where
  quotenvar : quoten (var x) ↓ (var x)
  quotenapp : quoten f ↓ m → quote v ↓ n → quoten (app f v) ↓ (app m n)

```

A coherence result in the style of Lemma 1 is proved by induction on the relation.

► **Lemma 3.**

$$\frac{\text{quote } v \downarrow n}{\ulcorner n \urcorner \equiv \ulcorner v \urcorner} \quad \frac{\text{quoten } m \downarrow n}{\ulcorner n \urcorner \equiv \ulcorner m \urcorner}$$

5.5 Normalisation

Finally, terms are normalised by evaluating in the identity environment and applying `quote`.

$$\text{norm } u \downarrow n = \Sigma(v : \text{Val } \Gamma A) \text{ eval } u \text{ idenv} \downarrow v \wedge \text{quote } v \downarrow n$$

With this definition, stability and completeness of BSN can already be proved.

4:12 Big Step Normalisation for Type Theory

► **Theorem 4** (Completeness).

$$\frac{\text{norm } u \Downarrow n}{\ulcorner n \urcorner \equiv u}$$

Proof. Immediate by Lemmas 1 and 3. ◀

► **Theorem 5** (Stability).

$$\frac{n : \text{Nf } \Gamma A}{\text{norm } \ulcorner n \urcorner \Downarrow n} \quad \frac{n : \text{NN } \Gamma A}{\Sigma(v : \text{NV } \Gamma A) \text{ eval } \ulcorner n \urcorner \Downarrow (\text{neu } v) \wedge \text{quoten } v \Downarrow n}$$

Proof. By simultaneous induction on normal forms and neutral normal forms. ◀

6 Correctness of BSN

Two main results must be proved in order to establish the correctness of BSN. Termination states that the normalisation relation is defined on every term.

$$\forall(u : \text{Tm } \Gamma A), \exists(n : \text{Nf } \Gamma A), \text{norm } u \Downarrow n$$

Soundness states that normalisation can only give one result for each term.

$$\frac{\text{norm } u \Downarrow n \quad \text{norm } u \Downarrow m}{n \equiv m}$$

Termination and soundness together imply that the normalisation relation defines a function from terms to normal forms, and the remaining coherence properties (completeness and stability) have already been proved in the previous section.

In this section, we first provide a short proof of soundness using known results on NBE.

Next, we define a partial normalisation of types, and the notion of skeleton of a type. Together, they give a very simple induction principle for the syntax of dependent types. Using this simplified induction principle, it is fairly straightforward to adapt the proof of termination for simple types [3], based on the strong computability predicate.

6.1 Soundness, by NBE

The original presentation of BSN for the simply-typed lambda-calculus proves soundness using a logical binary relation, similar to the use of strong computability for termination presented later in this section. Unfortunately, this proof seems hard to adapt to the quotiented syntax.

However there is an alternative proof, much shorter if not as interesting. The key observation is that BSN uses the same notion of normal forms as normalisation by evaluation (cf. [7] for a formal proof of NBE for type theory – we use the very same syntax and definition of normal forms). A direct consequence of the existence of a normalisation function such as NBE is that there is exactly one normal form in each equivalence class of terms, which in the quotiented syntax means that the embedding of normal forms is injective.

► **Theorem 6.**

$$\frac{n, m : \text{Nf } \Gamma A \quad \ulcorner n \urcorner \equiv \ulcorner m \urcorner}{n \equiv m}$$

Proof. By soundness and stability of normalisation by evaluation. ◀

► **Theorem 7** (Soundness).

$$\frac{\text{norm } u \Downarrow n \quad \text{norm } u \Downarrow m}{n \equiv m}$$

Proof. Immediate by Theorems 4 and 6. \blacktriangleleft

It can of course be argued that defining a normalisation function using another normalisation function defeats the object. However we think that it is interesting to consider BSN not so much as alternative normalisation function than as an alternative definition for the function which can also be obtained through NBE. This proof of soundness becomes more sensible from this point of view: as soon as we prove that the functions defined by NBE and BSN coincide (for which completeness of BSN is a key result), all correctness properties which are known to hold for NBE – in particular soundness – transfer to BSN.

6.2 Substitution-Free Types

An interesting issue was mentioned while defining `quote`: the natural definition is by induction on types, but only considers the constructors `U`, `El`, and `II`, forgetting both `_[]` and the quotient constructors. In this subsection, we show that this type of definition is in fact always correct, by defining substitution-free types, and proving that they are isomorphic to regular types.

Substitution free types are defined together with their embedding into regular types.

$$\begin{array}{ll}
 \mathbf{data} \text{ Ty}^{\text{sf}} : \text{Con} \rightarrow \text{Set} \text{ where} & \ulcorner _ \urcorner : \text{Ty}^{\text{sf}} \Gamma \rightarrow \text{Ty} \Gamma \\
 \text{U} : \text{Ty}^{\text{sf}} \Gamma & \ulcorner \text{U} \urcorner = \text{U} \\
 \text{El} : \text{Tm} \Gamma \text{ U} \rightarrow \text{Ty}^{\text{sf}} \Gamma & \ulcorner \text{El } u \urcorner = \text{El } u \\
 \text{II} : (A : \text{Ty}^{\text{sf}} \Gamma) \rightarrow \text{Ty}^{\text{sf}} (\Gamma, \ulcorner A \urcorner) \rightarrow \text{Ty}^{\text{sf}} \Gamma & \ulcorner \text{II } A \text{ B} \urcorner = \text{II } \ulcorner A \urcorner \ulcorner B \urcorner
 \end{array}$$

We will now define an evaluation function from types to substitution-free types, which will be the inverse of the embedding $\ulcorner _ \urcorner$. This requires to interpret every remaining type constructors in substitution-free types.

First, the application of a substitution to a substitution-free type is defined inductively.

$$\begin{array}{l}
 _[\sigma] : \text{Ty}^{\text{sf}} \Delta \rightarrow \text{Sub} \Gamma \Delta \rightarrow \text{Ty}^{\text{sf}} \Gamma \\
 \text{U}[\sigma] = \text{U} \\
 (\text{El } u)[\sigma] = \text{El}(u[\sigma]) \\
 (\text{II } A \text{ B})[\sigma] = \text{II} (A[\sigma]) (B[\sigma \uparrow \ulcorner A \urcorner])
 \end{array}$$

The definition directly follows the equations `U[]`, `El[]`, and `II[]` from the syntax of regular types. The remaining equations can be proved by induction.

$$\frac{A : \text{Ty}^{\text{sf}} \Gamma}{A[\text{id}] \equiv A} \qquad \frac{A : \text{Ty}^{\text{sf}} \Theta \quad \sigma : \text{Sub} \Delta \Theta \quad \nu : \text{Sub} \Gamma \Delta}{A[\sigma \circ \nu] \equiv A[\sigma][\nu]}$$

Put together, this defines the evaluation function: `U`, `El`, and `II` are interpreted by the respective constructors, substitutions are applied using the previous recursive definition, the equations `U[]`, `El[]`, and `II[]` hold trivially, and we just verified that `[id]` and `[]` are respected. It is easy to verify that this evaluation function is indeed the inverse of the embedding, therefore regular and substitution-free types are isomorphic.

This gives an alternative, much simpler induction principle for types.

► **Lemma 8.** *To define a function on types, it suffice to define it inductively for the constructors `U`, `El`, and `II`.*

Proof. The hypothesis of the lemma corresponds exactly to a definition of the function on substitution-free types. This function is then extended to regular types through the isomorphism previously defined. \blacktriangleleft

6.3 Type Skeletons

If we were to immediately define strong computability, we would face a second issue regarding the induction principle for types: it will often be the case that when proving a result by induction on types and considering a type $\Pi A B$, we need to apply the induction hypothesis not on B , but instead on $B[\sigma]$ for some substitution σ , which is not allowed by the induction principle of types. However, if we were to forget substitutions altogether, then B or $B[\sigma]$ would be the same. This is exactly the idea behind the skeleton of a type: by deleting all substitutions, we obtain a well-founded notion of size of types, for which B and $B[\sigma]$ are equivalent.

Formally, a type skeleton correspond to the non-dependent structure of types: either a base type or a function type.

```

data Sk : Set where
  base : Sk
   $\Pi$    : Sk  $\rightarrow$  Sk  $\rightarrow$  Sk

```

Defining the skeleton of a type is straightforward, and all quotient constructors are clearly respected.

```

skeleton : Ty  $\Gamma$   $\rightarrow$  Sk
skeleton U       = base
skeleton (El  $u$ ) = base
skeleton ( $\Pi$   $A$   $B$ ) =  $\Pi$  (skeleton  $A$ ) (skeleton  $B$ )
skeleton ( $A[\sigma]$ ) = skeleton  $A$ 

```

Using the skeleton of types as size indicators for induction, the example of problematic induction given at the beginning of this subsection becomes valid.

- **Lemma 9.** *To define a function f on types, it suffice to*
- *Define f on the base types U and El .*
 - *Define f on any type $\Pi A B$, while assuming that f is defined on C for any type C with the same skeleton as either A or B .*

Proof. The proof is the same as for Lemma 8, but additionally uses the skeletons as size indicators to ensure that the inductive definition is well-founded. Formally, this means that the function is defined by induction on type skeletons, then by pattern matching on the types of a given skeleton. ◀

6.4 Strong Computability

The proof of termination is based on a Tait-style [25] predicate on values, called strong computability. This subsection introduces strong computability, together with some important lemmas.

Strong computability is defined by induction on types, using Lemma 9

- A value v of a base type is SC if `quote` terminates on v .
- A value f of type $\Pi A B$ is SC if the application of f to a SC value v of type A gives a SC result of type B .

$$\begin{aligned}
\text{scv} &: \{A : \text{Ty}\} \rightarrow \text{Val } \Gamma \ A \rightarrow \text{Set} \\
\text{scv}_U \ v &= \Sigma(n : \text{Nf } \Gamma \ U) \ \text{quote } v \Downarrow n \\
\text{scv}_{(\text{El } u)} \ v &= \Sigma(n : \text{Nf } \Gamma \ (\text{El } u)) \ \text{quote } v \Downarrow n \\
\text{scv}_{(\Pi \ A \ B)} \ f &= \forall(\alpha : \text{Wk } \Delta \ \Gamma)(v : \text{Val } \Delta \ A^{+\alpha}) \rightarrow \text{scv } v \rightarrow \\
&\quad \Sigma(C : \text{Ty } \Delta) \ \Sigma(w : \text{Val } \Delta \ C) \\
&\quad (f^{+\alpha} \ @ \ v \Downarrow w) \ \wedge \ (\text{scv } w) \ \wedge \ (\text{skeleton } C \equiv \text{skeleton } B)
\end{aligned}$$

Some remarks can be made regarding the case of function types. Firstly, stability under application is understood up to weakening, i.e. the argument v need not be in the same context Γ as the function f , but may instead come from a weaker context Δ , where the weakening $\alpha : \text{Wk } \Delta \ \Gamma$ expresses that Δ is weaker than Γ .

Secondly, as in the definition of the evaluation relation, we prefer not to restrict the result type to simplify the upcoming proofs, hence we merely require that there exist a value w of some type C . However, the definition would not be well-founded without any restriction on C , since we inductively refer to strong computability at type C . Thus, we ask for C to have the same skeleton as B . In this way, strong computability for $\Pi \ A \ B$ is defined based on strong computability for types with the same skeleton as either A or B .

Strong computability is extended to environments pointwise.

$$\begin{aligned}
\text{sce} &: \text{Env } \Gamma \ \Delta \rightarrow \text{Set} \\
\text{sce } \epsilon &= \top \\
\text{sce } (\rho, v) &= \text{sce } \rho \ \wedge \ \text{scv } v
\end{aligned}$$

Let us now prove some lemma on strong computability. Throughout this subsection, we implicitly use Lemma 9 when proceeding by induction on types.

► **Lemma 10.** *Strong computability is stable under weakening:*

$$\frac{v : \text{Val } \Gamma \ A \quad \text{scv } v \quad \alpha : \text{Wk } \Delta \ \Gamma}{\text{scv } v^{+\alpha}} \quad \frac{\rho : \text{Env } \Gamma \ \Theta \quad \text{sce } \rho \quad \alpha : \text{Wk } \Delta \ \Gamma}{\text{sce } \rho^{+\alpha}}$$

Proof. For values, the proof is by induction on the type. For base types, stability of `quote` under weakening is used. For function types, the proof is immediate, since the definition of strong computability already accounts for weakening.

For environments, the proof is trivial by induction. ◀

► **Lemma 11.** *Strong computability is a mere proposition, i.e. any two proofs of strong computability are equal.*

$$\frac{p, q : \text{scv } v}{p \equiv q} \quad \frac{p, q : \text{sce } \rho}{p \equiv q}$$

Proof. For values, the proof is by induction on the type. For base types, we use soundness of `quote`, that is

$$\frac{\text{quote } v \Downarrow n \quad \text{quote } v \Downarrow m}{n \equiv m}$$

which follows easily from Lemma 3 and Theorem 6. For function types, Lemma 2 is used.

For environments, the proof is trivial by induction. ◀

The most important lemma regarding strong computability is that it implies termination of `quote`. A form of the converse for neutral values is proved simultaneously.

4:16 Big Step Normalisation for Type Theory

► **Lemma 12.**

$$\frac{v : \text{Val } \Gamma \ A \quad \text{sce } v}{\Sigma(n : \text{Nf } \Gamma \ A), \text{ quote } v \Downarrow n} \text{ (quote)}$$

$$\frac{v : \text{NV } \Gamma \ A \quad \Sigma(n : \text{NN } \Gamma \ A), \text{ quoten } v \Downarrow n}{\text{sce } (\text{neu } v)} \text{ (unquote)}$$

Proof. By mutual induction on the type A . The base cases are trivial by definition of strong computability. Consider a function type $\Pi A \ B$.

For the case *(quote)*, let f be a strongly computable value of type $\Pi A \ B$. Following the definition of **quote** for function types, we need to prove that there exist some $v : \text{Val } (\Gamma, A) \ B$ and $n : \text{Nf } (\Gamma, A) \ B$ such that

$$f^{+A} @ \text{neu } (\text{var } \text{vz}) \Downarrow v \quad \wedge \quad \text{quote } v \Downarrow n$$

In this expression, the variable vz has type $A[\pi_1 \text{id}]$. Furthermore **quoten** trivially terminates on variables, hence *(unquote)* implies that $\text{neu } (\text{var } \text{vz})$ is strongly computable by induction hypothesis. Then by definition of strong computability $f^{+A} @ \text{neu } (\text{var } \text{vz}) \Downarrow v$ holds for some strongly computable v , and we may verify using Lemma 1 that v has type B . Since v is strongly computable of type B , there exist by induction hypothesis $n : \text{Nf } (\Gamma, A) \ B$ such that $\text{quote } v \Downarrow n$. Therefore, $\text{quote } f \Downarrow (\lambda n)$.

Inversely, for the case *(unquote)*, assume $\text{quoten } f \Downarrow n$ with $f : \text{NV } \Gamma \ (\Pi A \ B)$, and let us prove that $\text{neu } f$ is strongly computable. Let $\alpha : \text{Wk } \Delta \ \Gamma$ and $v : \text{Val } \Delta \ A^{+\alpha}$ strongly computable. Let us prove that $\text{neu } (\text{app } f^{+\alpha} v)$ satisfies the conditions of the definition of strong computability for function types. Firstly,

$$(\text{neu } f^{+\alpha}) @ v \Downarrow (\text{neu } (\text{app } f^{+\alpha} v))$$

is immediate since f is neutral. Furthermore, by induction hypothesis *(unquote)* and definition of **quoten**, to prove that $\text{neu } (\text{app } f^{+\alpha} v)$ is strongly computable, it suffice to check that **quoten** terminates on $f^{+\alpha}$ and **quote** terminates on v . The former holds by hypothesis using that **quoten** is stable by weakening, while the latter holds by induction hypothesis *(quote)*. Finally, one may verify that the type of $\text{neu } (\text{app } f^{+\alpha} v)$ can be expressed as B with some substitutions and weakenings applied, hence its skeleton is the same as B . It follows that f is strongly computable. ◀

► **Lemma 13.** *The identity environment is strongly computable.*

$$\frac{\Gamma : \text{Con}}{\text{sce } \text{idenv}_\Gamma}$$

Proof. Lemma 12 implies that all variables are strongly computable because they are neutral values for which **quoten** trivially terminates. The result follows by induction on Γ , using Lemma 10. ◀

6.5 Termination

All the tools are now available to prove the main termination result.

► **Theorem 14.** *Evaluation in a strongly computable environment terminates, and yields a strongly computable result.*

$$\frac{u : \text{Tm } \Gamma \ A \quad \rho : \text{Env } \Delta \ \Gamma \quad \text{sce } \rho}{\Sigma(B : \text{Ty } \Delta) \Sigma(v : \text{Val } \Delta \ B) \text{ eval } u \ \rho \Downarrow v \quad \wedge \quad \text{sce } v}$$

$$\frac{\sigma : \text{Sub } \Gamma \ \Theta \quad \rho : \text{Env } \Delta \ \Gamma \quad \text{sce } \rho}{\Sigma(\nu : \text{Env } \Gamma \ \Theta) \text{ evals } \sigma \ \rho \Downarrow \nu \quad \wedge \quad \text{sce } \nu}$$

The theorem is proved by induction on terms and substitutions. Regular constructors are unproblematic, in the sense that the proofs does not change significantly compared to the case of an unquotiented syntax. However, we also need to verify that quotient constructors are respected, i.e. that for every equality constructor $u \equiv v$, the proof (seen as a function) of Theorem 14 gives equal results on u and v .

A simple way to ensure this is to prove that the types corresponding to Theorem 14 are mere propositions. In that case, when considering an equality constructor $u \equiv v$, the result of a proof of Theorem 14 on u and v will necessarily be equal since both are elements of the same mere proposition.

► **Lemma 15.** *For any $u : \text{Tm } \Gamma \ A$, $\sigma : \text{Sub } \Gamma \ \Theta$ and $\rho : \text{Env } \Delta \ \Gamma$, the following types are mere propositions.*

$$\begin{aligned} & \Sigma(B : \text{Ty } \Delta) \Sigma(v : \text{Val } \Delta \ B) \text{ eval } u \ \rho \Downarrow v \ \wedge \ \text{scv } v \\ & \Sigma(\nu : \text{Env } \Gamma \ \Theta) \text{ evals } \sigma \ \rho \Downarrow \nu \ \wedge \ \text{sce } \nu \end{aligned}$$

Proof. By Lemma 2, a term can only evaluate to a single value v . Furthermore, the types $\text{eval } u \ \rho \Downarrow v$ and $\text{scv } v$ are mere propositions, by definition and by Lemma 11 respectively. The result follows. The proof is similar in the case of substitutions. ◀

Proof of Theorem 14. By induction on terms and substitutions. We split the constructors into three groups:

- All quotient constructors are respected by Lemma 15.
- Almost all regular constructors are very straightforward: the result of evaluation is obtained by following the definition of the evaluator and applying the induction hypotheses, and strong computability of the result comes directly from the hypotheses. The exceptions to this pattern are λ and **app**, for which we give more detailed proofs below.
- For an abstraction λu of type $\Pi \ A \ B$ evaluated in a strongly computable environment $\rho : \text{Env } \Delta \ \Gamma$, evaluation is trivial since it simply yields the closure $\text{clos } u \ \rho$. Let us show that this closure is strongly computable.

Let $\alpha : \text{Wk } \Theta \ \Delta$, and $v : \text{Val } \Theta \ (A[\ulcorner \rho \urcorner]^{+\alpha})$ strongly computable. Then by Lemma 10, $(\rho^{+\alpha}, v)$ is a strongly computable environment, hence by induction hypothesis there exists w strongly computable such that $\text{eval } u \ (\rho^{+\alpha}, v) \Downarrow w$. It follows that $(\text{clos } u \ \rho)^{+\alpha} @ v \Downarrow w$. Finally, we may verify using Lemma 1 that the type of w must have the same skeleton as B . It follows that $\text{clos } u \ \rho$ is strongly computable.

- Consider an application **app** u with $u : \text{Tm } \Gamma \ (\Pi \ A \ B)$ evaluated in a strongly computable environment $(\rho, v) : \text{Env } \Delta \ (\Gamma, A)$. By induction hypothesis, there exists f strongly computable such that $\text{eval } u \ \rho \Downarrow f$. It can be verified using Lemma 1 that f has type $\Pi \ (A[\ulcorner \rho \urcorner]) \ (B[\ulcorner \rho \urcorner \uparrow A])$. Hence, because f and v are strongly computable, there exist w strongly computable such that $f @ v \Downarrow w$. Then we obtain by the definition of the evaluation relation that $\text{eval } (\text{app } u) \ (\rho, v) \Downarrow w$, proving the result. ◀

► **Theorem 16 (Termination).** *Normalisation terminates.*

$$\frac{u : \text{Tm } \Gamma \ A}{\Sigma(n : \text{Nf } \Gamma \ A), \text{ norm } u \Downarrow n}$$

Proof. Let $u : \text{Tm } \Gamma \ A$. By Lemma 13 and Theorem 14, there exist v strongly computable such that $\text{eval } u \ \text{idenv} \Downarrow v$. By Lemma 1, one may verify that v has type A . Finally, by Lemma 12, there exist $n : \text{Nf } \Gamma \ A$ such that $\text{quote } v \Downarrow n$. It follows that $\text{norm } u \Downarrow n$. ◀

By Theorems 7 and 16, `norm` defines a function from quotiented terms to normal forms, and by Theorems 4 and 5, it is the inverse of the embedding of normal forms. Therefore, we have proved that big step normalisation defines a normalisation function.

7 Formalisation of BSN in a Cubical Type Theory

Most of the present work has been formalised [5] using Agda [23]. Precisely, Sections 3 to 5 have been fully formalised, and Section 6 partially so – what remains to do in the latter is equality reasoning. This formalisation is expressed in a cubical type theory (CTT, cf. [11]) using the cubical mode of Agda. This differs from the present paper, which uses a strict type theory for simplicity. The choice of CTT allows to easily express QIIT as a special case of higher inductive types (HIT, cf. [24]), which from the technical point of view is a notable improvement over previous implementations of QIIT in non-cubical Agda, which had to introduce all quotient constructors as additional axioms (e.g. [7, 8]).

As explained in [8], simply considering a QIIT as a special case of HIT leads to unexpected results. For instance, in the case of the quotiented syntax, `U[]` and `[id]` give two proofs of `U[id] ≡ U`, and these proofs are distinct in a non-strict type theory. Therefore, this naive implementation of QIIT leads to a syntax which is not a set in the type theoretic sense, i.e. uniqueness of identity proofs (UIP) does not hold. It follows by Hedberg’s theorem [17] that equality is undecidable in this syntax, which is definitively not what was expected.

The solution is to truncate the syntax to a set, by the addition of the following constructors:

$$\begin{aligned} \text{setTy} & : \{A B : \text{Ty } \Gamma\} (p q : A \equiv B) \rightarrow p \equiv q \\ \text{setSub} & : \{\sigma \nu : \text{Sub } \Gamma \Delta\} (p q : \sigma \equiv \nu) \rightarrow p \equiv q \\ \text{setTm} & : \{s t : \text{Tm } \Gamma A\} (p q : s \equiv t) \rightarrow p \equiv q \end{aligned}$$

Note that the corresponding constructor for contexts is unnecessary, because it can be proved that contexts form a set from the fact that types are a family of sets.

In order to adapt the proof of big step normalisation to CTT with this implementation of QIIT, there are two problems to solve:

- The proof of BSN uses the UIP axiom of the strict type theory.⁴ Since we lose this axiom in CTT, its uses must be replaced.
- The additional truncation constructors of QIIT must be taken into account whenever we use induction on a QIIT.

Both problems can be solved together by proving that all the types considered in the proof of BSN are in fact sets. Indeed, any use of the UIP axiom can then be replaced by the proof of UIP for the appropriate type, and when defining a function by induction on a QIIT, the set-truncation constructor can be mapped to the proof that the codomain is a set.

We will not detail all the proofs of UIP because they are fairly repetitive, but we will explain the general techniques used. There are some trivial cases: all QIIT (terms, types, values) are explicitly truncated to sets. Mere propositions (the big step relations, strong computability) are always sets. What remains are regular inductive types, such as variables, normal forms, substitution-free types... For such types, Hedberg’s theorem [17] is an

⁴ While we never explicitly refer to the UIP axiom in this presentation of the proof, it is used whenever we prove a lemma of the form $L : (x : A) \rightarrow f(x) \equiv g(x)$ by induction on a QIIT A . Indeed, for a quotient constructor of type $a \equiv b$ in A , we need to provide an equality between equalities $L(a) \equiv L(b)$. This is trivial with UIP – hence why such cases are neglected in the proof – but is in general problematic in a non-strict metatheory. An example is the coherence lemmas for weakening of values.

extremely useful tool. For instance, it is easy to verify that equality of variables is decidable, which implies that they are a set. Even for types which do not a priori have decidable equality (e.g. normal forms), it is still possible to adapt the techniques and lemmas used for Hedberg’s theorem to prove UIP.

8 Conclusion and Further Work

We have formalised big step normalisation for a simple dependent type theory, and proved its correctness. Crucially, a quotiented syntax of type theory based on QIIT is used to reduce the complexity of this proof. While the proof of BSN for type theory shares many similarities with the case of the simply-typed lambda-calculus, it requires some additional steps, for instance a simplified induction principle for the syntax of types.

This work is also an interesting application of the QIIT syntax of type theory, since it provides an example in which using this syntax has an important impact on the proof. The implementation of the QIIT syntax using HIT in cubical Agda, and its use in the formalisation of BSN is also a practical validation of ideas which were developed in [8].

Since we have only considered a minimalist type theory in this work, it is natural to try to extend it. A first step in this direction could be the addition of some inductive types. This was already done in the non-dependently typed case in [3], by adding integers (System T). In order to handle inductive types in BSN, the general idea is to add the inductive constructors to values and normal forms, and add the elimination principles to neutral values and neutral normal forms, adapting `eval` and `quote` accordingly. A next step could then be to add W -types, so as to allow the use of arbitrary inductive types. Another equally interesting extension would be to replace the abstract universe U – which contains no closed terms – with a more useful universe equipped with type constructors.

References

- 1 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 293–310. Springer, Cham, 2018.
- 2 Thorsten Altenkirch and James Chapman. Tait in one big step. In *MSFP@ MPC*, 2006.
- 3 Thorsten Altenkirch and James Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.
- 4 Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310. IEEE, 2001.
- 5 Thorsten Altenkirch and Colin Geniet. Agda formalisation for the paper big step normalisation for type theory. Available at <https://github.com/colingeniet/big-step-normalisation>, 2019.
- 6 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *International Conference on Category Theory and Computer Science*, pages 182–199. Springer, 1995.
- 7 Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for dependent types. In *1st conference on Foundational Structures in Computation and Deduction (FSCD)*, 2016.
- 8 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, 2016.
- 9 Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE, 1991.

- 10 James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009.
- 11 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint*, 2016. [arXiv:1611.02108](https://arxiv.org/abs/1611.02108).
- 12 Thierry Coquand. An algorithm for testing conversion in type theory. *Logical frameworks*, 1:255–279, 1991.
- 13 Gabe Dijkstra. *Quotient inductive-inductive definitions*. PhD thesis, University of Nottingham, 2017.
- 14 Daniel J Dougherty and Ramesh Subrahmanyam. Equality between functionals in the presence of coproducts. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 282–291. IEEE, 1995.
- 15 Peter Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995.
- 16 Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.
- 17 Michael Hedberg. A coherence theorem for martin-löf’s type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.
- 18 Martin Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997.
- 19 Jean-Pierre Jouannaud and Albert Rubio. Rewrite orderings for higher-order terms in η -long β -normal form and the recursive path ordering. *Theoretical Computer Science*, 208(1-2):33–58, 1998.
- 20 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):2, 2019.
- 21 Paul Blain Levy. *Call-by-push-value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.
- 22 Paul-André Mellies. Typed λ -calculi with explicit substitutions may not terminate. In *International Conference on Typed Lambda Calculi and Applications*, page 32. Springer, 1995.
- 23 Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.
- 24 Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. Univalent Foundations, 2013.
- 25 W. W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967. [doi:10.2307/2271658](https://doi.org/10.2307/2271658).

From Cubes to Twisted Cubes via Graph Morphisms in Type Theory

Gun Pinyo 

School of Computer Science, University of Nottingham, UK
gunpinyo@gmail.com

Nicolai Kraus 

School of Computer Science, University of Nottingham, UK
<https://nicolaikraus.github.io/>
nicolai.kraus@nottingham.ac.uk

Abstract

Cube categories are used to encode higher-dimensional categorical structures. They have recently gained significant attention in the community of homotopy type theory and univalent foundations, where types carry the structure of higher groupoids. Bezem, Coquand, and Huber [8] have presented a constructive model of univalence using a specific cube category, which we call the *BCH cube category*.

The higher categories encoded with the BCH cube category have the property that all morphisms are invertible, mirroring the fact that equality is symmetric. This might not always be desirable: the field of *directed type theory* considers a notion of equality that is not necessarily invertible.

This motivates us to suggest a category of *twisted cubes* which avoids built-in invertibility. Our strategy is to first develop several alternative (but equivalent) presentations of the BCH cube category using morphisms between suitably defined graphs. Starting from there, a minor modification allows us to define our category of twisted cubes. We prove several first results about this category, and our work suggests that twisted cubes combine properties of cubes with properties of globes and simplices (tetrahedra).

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases homotopy type theory, cubical sets, directed equality, graph morphisms

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.5

Related Version This paper is also available at <https://arxiv.org/abs/1902.10820>.

Funding *Nicolai Kraus*: The Royal Society, grant No. URF\R1\191055.

Acknowledgements We would like to thank Paolo Capriotti and Jakob von Raumer. Both offered many suggestions during fruitful exchanges. In particular, the initial observation on which Theorem 21 is based was suggested by them, and the idea of considering graph morphisms was found in one of our many interesting discussions. We are also grateful to the participants of TYPES'19 in Oslo and the summer school on HTT/UF in Leeds. We thank in particular Emily Riehl, Christian Sattler, and Steve Awodey for their help and their comments. Special thanks go to Andreas Nuyts, who has pointed out a mistake in an earlier draft of this paper, and to the anonymous reviewers for their careful reading and comments.

1 Introduction and Motivation

A *cube category* is a category whose objects are (or represent) finite-dimensional cubes, and whose morphisms are mappings of some sort between these cubes. There are many different cube categories [1, 5, 8, 9, 20], and they are used to encode higher categorical structures.



© Gun Pinyo and Nicolai Kraus;

licensed under Creative Commons License CC-BY

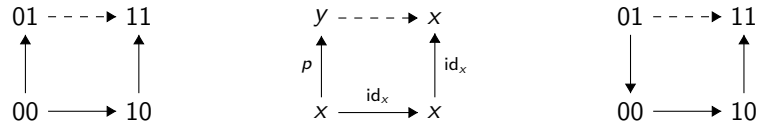
25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 5; pp. 5:1–5:18

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Kan-filling condition of a 2-cube (left), a proof of invertibility introduced by the Kan-filling condition (middle), and how to remove such the invertibility (right).

Homotopy type theory [28] is a variation of Martin-Löf’s intensional type theory. The characteristic and novel view adapted in homotopy type theory is that types carry the structure of higher categories, or, to be precise, higher groupoids (i.e. all morphisms are invertible). This view supports Voevodsky’s *univalence principle* which should be seen as a central concept of homotopy type theory. The first model of such a type theory, given by Voevodsky [29] (see also the presentation by Kapulkin and Lumsdaine [16]), uses *simplicial sets*. However, it is still an open question how simplicial sets can be used to build a *constructive* model of type theory with univalent universes [13]. Using *cubical sets*, this has been achieved by Bezem, Coquand, and Huber [8]. Starting from there, cubes have gathered a lot of attention in the type theory community, leading to various *cubical type theories* which have univalence not as an axiom but as a built-in derivable principle [3, 6, 12, 23]. Many different cube categories have been considered in this context.

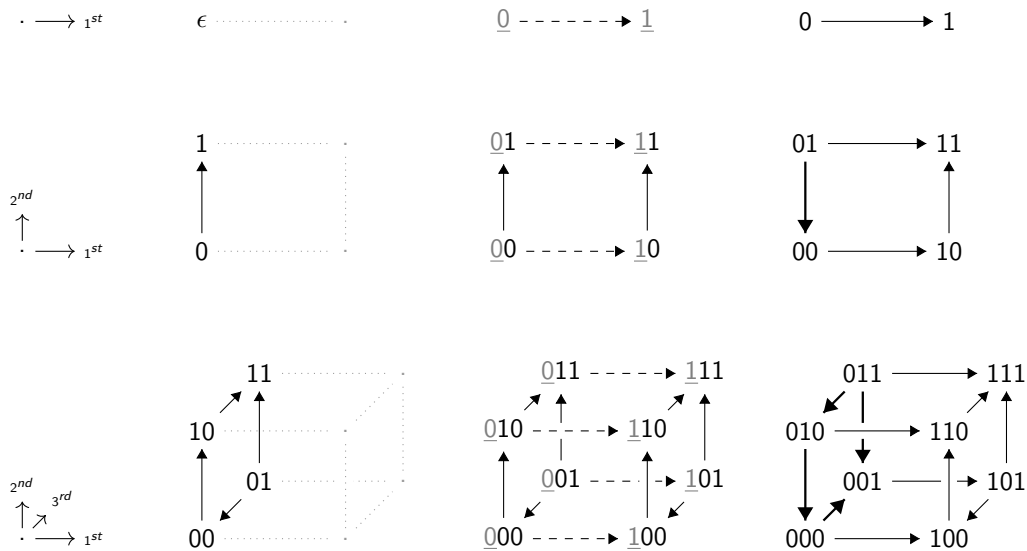
The important cube category used by Bezem, Coquand, and Huber [8] (from now on referred to as the *BCH cube category*) uses finite sets of variable names as objects, and a morphism from a set I to a set J is a function $f : I \rightarrow J \cup \{0, 1\}$ which is “injective on the left part”, i.e. $f(i_1) = f(i_2) = j$ with $j : J$ implies $i_1 = i_2$. One goal of this paper is to develop several alternative presentations of this category, mainly using graph morphisms. We have two main motivations to do this. The first is that, as we hope, our alternative and intuitive (but equivalent) definitions enable new views on the category and facilitate the discovery of further observations. The second motivation is that a minor change in the definition will allow us to construct a new cube category, the *twisted cubes* from the title. We will come back to this in a moment.

The standard way to create models (of both higher categories and type theories) using simplicial or cubical index categories is to take presheaves and equip them with certain *Kan-filling conditions*. These filling conditions entail composition of morphisms as well as associativity and all higher coherence laws that one needs. A typical such Kan-filling condition for the 2-cube¹, as shown on the left of Figure 1, says that, given the “partial square” of three solid edges on the right, one can always find the dashed edge (together with an actual filler for the square).

One important observation here is that, in the case of the BCH cube category and other cube categories, invertibility of morphisms is built-in. Consider the partial square, as shown on the middle of Figure 1, where two of the three solid edges are identities and the third is an actual non-trivial morphism (or equality) p from x to y . Using the Kan filling operation described above, we get a morphism from y to x , which serves as the inverse of p .

The invertibility of morphisms is useful for most forms of type theory, where equality is symmetric. This however is not always the case, cf. the proposals for *directed type theories* by Licata and Harper [18], Nuyts [22], Riehl and Shulman [25], North [21], and others. Their

¹ While Bezem, Coquand, and Huber [8] define their index category to have finite sets of variables as objects, it is possible to simply use natural numbers as objects. The *n-cube*, or *n-dimensional cube*, is then the object of the presheaf category that is represented by the object n of the index category.



■ **Figure 2** An illustration of the *thickening-and-twisting* process of the twisted n -cube for $1 \leq n \leq 3$. The process expands the twisted $(n - 1)$ -cube (left column) along the new dimension (middle column) and reverse all other dimensions at the starting point of the new dimension (right column).

aim is to generalise type theory by replacing (*higher*) *groupoids* by general (*higher*) *categories*. In a nutshell, this means that “equality” (or whatever takes the place of equality) is not necessarily invertible.

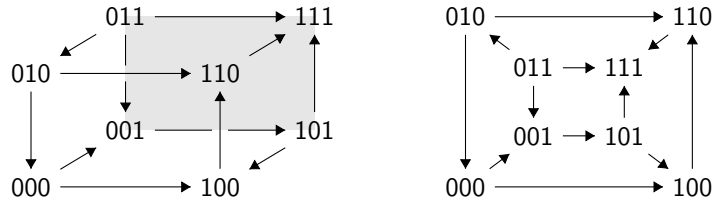
We think that a very valuable long-term goal would be to make the connection of directed type theories with cubical type theories and create some sort of *directed cubical type theory*. This is at the moment certainly out of reach, and we do not know how such a type theory could be built. Nevertheless, it motivates us to explore variations of the BCH cube category which do not have the described built-in equality.

To avoid invertibility, we “twist” the left-most edge of the 2-dimensional cube, as shown on the right of Figure 1, to ensure that the construction from before becomes impossible. This might seem artificial and specific to the 2-dimensional case but by using our graph morphisms that we develop for the BCH cube category, it becomes very easy to define the twisting version for cubes of all dimensions.

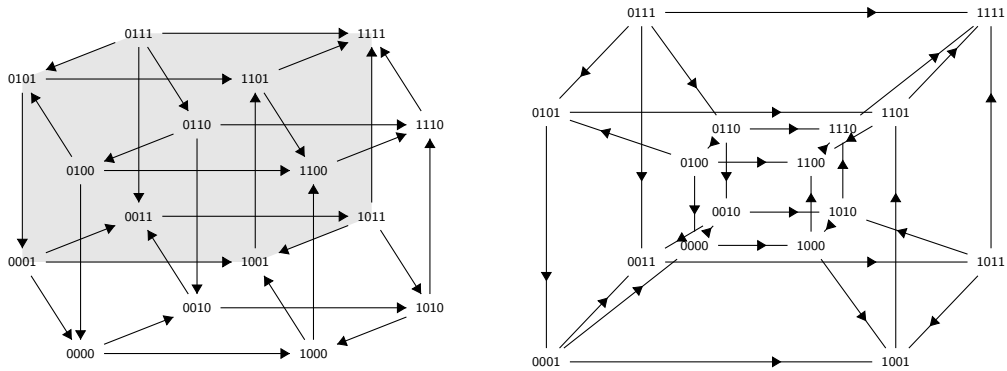
To construct a twisted n -cube from a twisted $(n - 1)$ -cube, we first expand the original cube along a new dimension (we call this *thickening*): this is same as constructing a standard n -cube from a standard $(n - 1)$ -cube, which is just a construction of its *cylinder object*. We then reverse all dimensions at the starting point of the new dimension (we call this *twisting*). Figure 2 illustrates this *thickening-and-twisting* process up to dimension 3, where the existing dimensions are shifted by one in order to allow the new dimension to be the first dimension.

One important property of standard cubes which twisted cubes retain is that every face of a [twisted] n -cube is a [twisted] $(n - 1)$ -cube. An interesting example is the case $n = 3$: In order to construct a twisted 3-cube, we thicken the twisted 2-cube as illustrate in Figure 2 where the left and the right face are already twisted 2-cubes, while the rest are thickened 1-cubes. The right face is unaffected during the twisting, but the left face is reversed entirely. Nevertheless, it is still a 2-cube (just flipped backwards).

5:4 From Cubes to Twisted Cubes via Graph Morphisms in Type Theory



■ **Figure 3** The 3-dimensional twisted cube using parallel and perspective projections. On the left, the lid (i.e. the last face which can be recovered by filling) is shaded. On the right, this face is the small middle square. The lid can be seen as the composite of the other faces.



■ **Figure 4** The 4-dimensional twisted cube using parallel and perspective projections. The lid is shadowed on the left. It is the biggest cube on the right.

Twisted cubes do not only remove the discussed source of invertibility, but they also change the way we view composition of morphisms. The filling of a “standard” square can be interpreted as saying that the composition of two edges equals the composition of the other two edges, and if we want to see the lid as the composite of the three other edges, then one has to be inverted. In contrast, in the twisted square, the lid can be seen directly as the single composite of the three other edges. The right half of Figure 3 shows the projection of the twisted 3-cube, and the smallest square (011, 001, 101, 111) is the lid. As for the square, this lid should be seen as the composite of the other (here five) faces. Intuitively, one starts with the biggest square, composes it with the top and the bottom squares, then with the left and the right square, and thus arrives at the smallest square. Figure 4 shows the analogous situation for the 4-dimensional twisted cube, where one starts with the inner 3-cube, then extends to the front and the back, to the top and the bottom, and finally to the left and the right.

The “twisting” pattern also appears in the *twisted arrow category* [17], also known as the *category of factorisations* [7]. However, it is unclear how to generalise this idea to more than squares; it has been developed to solve a different problem.

In the main body of the paper, we first introduce the framework of graph morphisms for standard (non-twisted) cubes. We consider the properties of meet/join and dimension preservation of graph morphisms, and conclude that both of these are suitable refinements to ensure that the category of graph morphisms matches the BCH cube category. The proof of this is the main result of Section 2. We use this development to introduce and examine *twisted cubes* in Section 3. We will see that they have many characteristic properties that

standard cubes are lacking. Some of them, such as a Hamiltonian path through the cube and the fact that vertices are totally ordered, are familiar from simplicial structures but not from cubical ones. Another interesting feature, neither familiar from cubical nor from simplicial but from globular structures, is that surjective maps are unique (i.e. there is only one way to degenerate a twisted cube). These and other observations allow us to define a further representation of the category of twisted cubes which does not make use of graphs.

Setting. We use a standard version of Martin-Löf’s dependent type theory as our meta-language. We assume function extensionality, but we do not require other axioms or features since we mostly work with finite sets, which are extremely well-behaved by default. In particular, it does not matter for us whether UIP/Axiom K is assumed or not, and the development would be identical in extensional dependent type theory.

Summary of Contributions. Our main contributions are as follows:

- We give several alternative but equivalent presentations of the BCH cube category.
- We introduce *twisted cubes*, a variation of the BCH cube category which allows for filling conditions without built-in invertibility.
- We show several results about twisted cubes. These include connections to simplices (a unique Humiliation path and the property of being a Reedy category) and to globes (unique surjective maps and degeneracies).

2 A Standard Cube Category

In this section, we discuss various representations of the cube category \square_{BCH} . This category was used by Bezem, Coquand, and Huber to present a constructive model of univalence [8]. In Section 3, we will see how minimal modifications lead to a category of twisted cubes.

Keeping in mind that we use type theory as the language in which the results are presented (i.e. as our meta-theory), we use the following notations: \mathbb{N} are the natural numbers, including 0. For $n : \mathbb{N}$, the set \underline{n} is the finite set with elements $\{0, 1, \dots, n - 1\}$. In particular, $\underline{2}$ is the set of booleans. As usual, $\underline{m}^{\underline{n}}$ is simply the function set $\underline{m} \rightarrow \underline{n}$. We denote elements of $\underline{2}^{\underline{n}}$ by binary sequences as in $0 \cdot 1 \cdot 1 \cdot 0$. This means such a function f is denoted by $f(0) \cdot f(1) \cdot f(2) \dots f(n - 1)$. If there is no risk of confusion, we omit the \cdot and simply use juxtaposition as in 0110.

In several situations, we want to consider a type of functions into a coproduct which is injective “on the *left* part of the codomain”. To make this precise, we introduce a notation:

► **Definition 1** ($\xrightarrow{\text{left}}$). Assume A , B , and C are given types. For a function $f : A \rightarrow (B + C)$, we say that f is injective on the left part if

$$\text{left-inj}(f) := \prod (x, y : A, z : B). (f(x) = \text{inl}(z)) \rightarrow (f(y) = \text{inl}(z)) \rightarrow x = y. \quad (1)$$

We write the type of functions which are injective on the left part as

$$(A \xrightarrow{\text{left}} B + C) := \Sigma (f : A \rightarrow (B + C)). \text{left-inj}(f). \quad (2)$$

In the next lemma, a function $f : A \rightarrow B + \underline{1}$ is called a *partial function*, with $\underline{1}$ being the “undefined” part.² The following simple but useful (and well-known) result will be necessary. It could be formulated in higher generality, but a version which is sufficient for us is this:

► **Lemma 2.** *Given $m, n : \mathbb{N}$, injective partial functions from \underline{m} to \underline{n} are in bijection with injective partial functions from \underline{n} to \underline{m} . In other words, we have an equivalence*

$$\left(\underline{m} \xrightarrow{\text{left}} \underline{n} + \underline{1} \right) \simeq \left(\underline{n} \xrightarrow{\text{left}} \underline{m} + \underline{1} \right). \quad (3)$$

Proof. The equivalence can be constructed directly. Given an $f : \underline{m} \xrightarrow{\text{left}} \underline{n} + \underline{1}$, we have to construct a function $g : \underline{n} \xrightarrow{\text{left}} \underline{m} + \underline{1}$. For $i : \underline{n}$, we can decide whether there is a k such that $f(k) = \text{inl}(i)$. If so, then this k is unique due to injectivity, and we set $g(i) := \text{inl}(k)$; otherwise, we set $g(i) := \text{inr}(0)$. Checking that this is an equivalence is routine. ◀

The presentation of the cube category in question that we start with is the one given by Bezem, Coquand, and Huber [8] (which is the same as in Huber’s PhD thesis [15]). Since it is sufficient for our purposes, we use a skeletal variation: our objects are not finite sets but rather natural numbers.

► **Definition 3** (category \square_{BCH} [8, 15]). *The category \square_{BCH} has natural numbers as objects and, for $m, n : \mathbb{N}$, a morphism in $\square_{\text{BCH}}(m, n)$ is a function $f : \underline{m} \rightarrow \underline{n} + \underline{2}$ which is injective on the \underline{n} -part. In type-theoretic notation:*

$$\text{obj}(\square_{\text{BCH}}) := \mathbb{N} \qquad \square_{\text{BCH}}(m, n) := \underline{m} \xrightarrow{\text{left}} \underline{n} + \underline{2} \quad (4)$$

Composition $g \circ f$ is defined to be the set-theoretic composition $(g + \text{id}_2) \circ f$.

What we will need is the opposite of this category, $\square_{\text{BCH}}^{\text{op}}$. While the above definition is short and abstract, a description close to the intuitive idea of cubes is helpful for our later developments. Let us consider *graphs* $G = (V, E)$ of nodes (vertices) and edges, where V is a set with decidable equality and E is a subset of $V \times V$. A standard way to implement this is to let E be a family of “mere propositions”³, indexed twice over V . However, we write $(s, t) : E$ for $E(s, t)$ and assume that E is given in the “total space” formulation. Furthermore, in our cases E will always be a *decidable* subset.

E being a subset means that our graphs do *not* have multiple parallel edges, i.e. for any pair of vertices, there is at most one edge between them, and it is decidable whether there is an edge between two given vertices.

Given a graph, we construct a new graph as follows. Note that the “total space” of the edges of the new graph is $E + E + V$, but in order to make clear which vertices these new edges connect, we use “set theory style” notation:

► **Definition 4.** *Given $G = (V, E)$, the graph-prism of G , denoted as $\text{prism}(G) := (\text{prism}(V), \text{prism}(E))$ is another graph where*

$$\text{prism}(V) := \underline{2} \times V \quad (5)$$

$$\text{prism}(E) := \{ ((0, s), (0, t)) \mid (s, t) : E \} \quad (6)$$

$$\cup \{ ((1, s), (1, t)) \mid (s, t) : E \} \quad (7)$$

$$\cup \{ ((0, v), (1, v)) \mid v : V \}. \quad (8)$$

² Technically, these are of course only the partial functions from A to B with decidable support. Since we only work with finite types, it is not surprising that we only need to consider the decidable case.

³ Recall that a *mere proposition*, or a *subsingleton*, is a type with at most one element.

This allows us to define the standard cube as a graph:⁴

► **Definition 5.** Given $n \in \mathbb{N}$, the standard cube C_n is defined as follows:

$$C_0 := (\underline{1}, \{(0, 0)\}) \qquad C_{n+1} := \text{prism}(C_n) \qquad (9)$$

Another way of defining C_n , without recursion, is the following. Here, we give the “total space” of edges $\text{edges}(C_n)$ together with functions $\text{src}, \text{trg} : \text{edges}(C_n) \rightarrow \text{nodes}(C_n)$:

► **Definition 6.** In the following, our convention is that $\underline{-1}$ is empty (i.e. the same as $\underline{0}$):

$$\text{nodes}(C_n) \qquad \qquad \qquad \equiv \qquad \qquad \underline{2}^n \qquad (10)$$

$$\text{edges}(C_n) \qquad \qquad \qquad \equiv \qquad \qquad \underline{2}^n + (\underline{n} \times \underline{2}^{n-1}) \qquad (11)$$

$$\text{src}(\text{inl}(v)) \quad \equiv \quad \text{trg}(\text{inl}(v)) \qquad \equiv \qquad v \qquad (12)$$

$$\text{src}(\text{inr}(i, x_0 x_1 \dots x_{n-2})) \qquad \equiv \qquad x_0 x_1 \dots x_{i-1} 0 x_i \dots x_{n-2} \qquad (13)$$

$$\text{trg}(\text{inr}(i, x_0 x_1 \dots x_{n-2})) \qquad \equiv \qquad x_0 x_1 \dots x_{i-1} 1 x_i \dots x_{n-2} \qquad (14)$$

The number of total edges in (11) comes from the following calculation. We have n dimension, thus 2^n nodes, which come with self-loops giving rise to the summand $\underline{2}^n$. For ever node, we further have an edge in each dimension. Avoiding double counting, this gives the summand $\underline{n} \times \underline{2}^{n-1}$. Figure 5 shows drawings for C_0 to C_3 .

► **Lemma 7.** Definition 5 and Definition 6 define isomorphic graph structures. ◀

This observation allows us to use whichever is more convenient in any given situation.

A *graph morphism* from $G = (V, E)$ to $G' = (V', E')$ is, as usual, a function between the node types which preserves the edges:

$$\text{grp-hom}((V, E), (V', E')) := \Sigma(f : V \rightarrow V'). \Pi(v_0, v_1 : V). E(v_0, v_1) \rightarrow E'(f(v_0), f(v_1)) \quad (15)$$

We can now consider the following category:

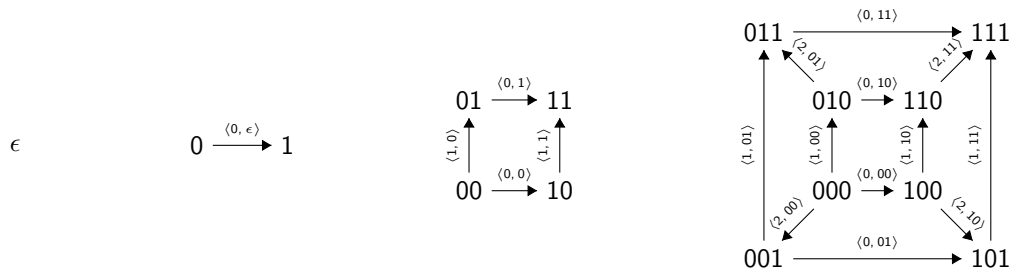
► **Definition 8** (category \square_{grp}). The category \square_{grp} has natural numbers as objects.

A *morphism* between m and n is a graph morphism from C_m to C_n , as in:

$$\text{obj}(\square_{\text{grp}}) := \mathbb{N} \qquad \square_{\text{grp}}(m, n) := \text{grp-hom}(C_m, C_n) \qquad (16)$$

Composition is composition of graph morphisms.

⁴ Most of graphs in this paper are reflexive graphs to support degeneracies as graph morphisms.



■ **Figure 5** An illustration of C_n for $n \leq 3$. The labels on the vertices and edges are in accordance with (10) and (11). The identity loops are omitted. This allows us to unambiguously hide the constructor inr as well.

The category \square_{grp} has more morphisms than $\square_{\text{BCH}}^{\text{op}}$. One example would be the morphism in $\text{grp-hom}(C_2, C_1)$ which maps the three nodes 00, 01, 10 all to 0 and 11 to 1. Another example is the morphism which maps 00 to 0, and 01, 10, 11 all to 1. Both of these morphisms do not have analogues in $\square_{\text{BCH}}^{\text{op}}$. In other words, \square_{grp} has *connections*. Since, in the current paper, we are looking for alternative definitions of the category $\square_{\text{BCH}}^{\text{op}}$, we refine the definition of the morphisms in \square_{grp} to resolve the mismatch. Let us formulate the following auxiliary definitions.

► **Definition 9** (free preorder on a graph). *For a given graph $G = (V, E)$, we write $G^* = (V, E^*)$ for the free preorder generated by it. G^* has V as objects and, for $v, u : V$, we have $v \leq u$ if there is a chain of edges starting in v and ending in u .*

When talking about nodes in G , we borrow the notions of meet (product) and join (coproduct) from preorders. If they exist in G^ , we write them as $v \sqcap u$ and $v \sqcup u$.*

It is easy to see that, in the case of C_n , all meets and joins exist and can be calculated directly: From the programming perspective, they correspond to the bitwise operators '&' and '|'. Thus, when talking about C_n , we can view \sqcap and \sqcup as actual functions calculating the binary meet and join:

$$\sqcap, \sqcup : V \times V \rightarrow V \quad (17)$$

Given a graph morphism $g : \text{grp-hom}(C_m, C_n)$, it is easy to define what it means that it preserves binary meets resp. joins:

$$\text{pres-meet}(g) := \Pi(u, v : \underline{2}^m). g(u \sqcap v) = g(u) \sqcap g(v) \quad (18)$$

$$\text{pres-join}(g) := \Pi(u, v : \underline{2}^m). g(u \sqcup v) = g(u) \sqcup g(v) \quad (19)$$

Note that preserving meets and joins is a property (a “mere proposition”) of morphisms. For general morphisms between graphs which might not have all meets or joins, the definition is more subtle but still straightforward; one can always define the property of *being a meet (join)* and then say that any vertex which has this property is mapped to one which also has it. We omit the precise type-theoretic formulation.

The two mentioned examples of morphisms which are “too much” in \square_{grp} do not preserve binary meets resp. joins.

► **Definition 10** (category \square_{cont}). *The category \square_{cont} has \mathbb{N} as objects and, as morphisms, graph morphisms between standard cubes which preserve meets and joins (cont for continuous):*

$$\text{obj}(\square_{\text{cont}}) := \mathbb{N} \quad (20)$$

$$\square_{\text{cont}}(m, n) := \Sigma(g : \text{grp-hom}(C_m, C_n)). \text{pres-meet}(g) \times \text{pres-join}(g) \quad (21)$$

This gives us a category which is indeed equivalent (in fact isomorphic) to $\square_{\text{BCH}}^{\text{op}}$:

► **Theorem 11.** *The categories $\square_{\text{BCH}}^{\text{op}}$ and \square_{cont} are isomorphic. The isomorphism on the object part is the identity, i.e. the equivalence is given by a family e as in:*

$$e : \Pi(m, n : \mathbb{N}). \square_{\text{BCH}}^{\text{op}}(m, n) \simeq \square_{\text{cont}}(m, n). \quad (22)$$

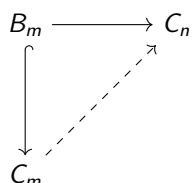
Before giving a proof, we formulate the following:

► **Lemma 12.** *Consider the full subgraph of C_n which has exactly $(n + 1)$ vertices, namely the “origin” $00 \dots 0$ and the “base vectors” which have exactly one 1. We call this subgraph B_n , where the B stands for “base”, and it comes with the inclusion $i : B_n \hookrightarrow C_n$. For any m , “forgetting” the property of preserving the joins and composing with i as in*

$$\lambda g. i \circ (\text{proj}_1(g)) : (\Sigma(g : \text{grp-hom}(C_m, C_n)). \text{pres-join}(g)) \rightarrow \text{grp-hom}(B_m, C_n) \quad (23)$$

is an equivalence. Moreover, g preserves meets if and only if $i \circ (\text{proj}_1(g))$ does.

Proof. The only binary joins that B_m has are trivial, so every morphism $\text{grp-hom}(B_m, C_n)$ is join-preserving. Thus, the first claim of the lemma is that every such morphism can be extended in a unique way as shown in the diagram to the right. Every node of C_m which is not in B_m , i.e. every node which is not the origin or a base vector, can be written as a join of base vectors. Since we need to preserve joins, it is therefore determined where the node has to be sent to. The map defined in this way preserves all binary joins, and it preserves binary meets if and only if the input does. ◀



Proof of Theorem 11. We first give the overview of the argument as a chain of equivalences, then we justify each step [S1 – S5].

$$\begin{aligned}
& \square_{\text{cont}}(m, n) \\
& \equiv \Sigma(g : \text{grp-hom}(C_m, C_n)).\text{pres-meet}(g) \times \text{pres-join}(g) \\
\text{[S1]} & \simeq \Sigma(g : \text{grp-hom}(B_m, C_n)).\text{pres-meet}(g) \\
\text{[S2]} & \simeq \Sigma(z : \underline{2}^{\underline{2}}, d : \underline{m} \xrightarrow{\text{left}} \underline{n} + \underline{1}).\Pi(i : \underline{m}, j : \underline{n}).(d(i) = \text{inl}(j)) \rightarrow (z(j) = 0) \\
\text{[S3]} & \simeq \Sigma(z : \underline{2}^{\underline{2}}, e : \underline{n} \xrightarrow{\text{left}} \underline{m} + \underline{1}).\Pi(i : \underline{m}, j : \underline{n}).(e(j) = \text{inl}(i)) \rightarrow (z(j) = 0) \\
\text{[S4]} & \simeq \Sigma(z : \underline{2}^{\underline{2}}, e : \underline{n} \rightarrow (\underline{m} + \underline{1})).\text{left-inj}(e) \times \Pi(i : \underline{m}, j : \underline{n}).(e(j) = \text{inl}(i)) \rightarrow (z(j) = 0) \\
\text{[S5]} & \simeq \Sigma(\alpha : \Pi(j : \underline{n}).\Sigma(e : \underline{m} + \underline{1}, z : \underline{2}).\Pi(i : \underline{m}).(e = \text{inl}(i)) \rightarrow z = 0).\text{left-inj}(\text{proj}_1 \circ \alpha) \\
\text{[S6]} & \simeq \Sigma(\alpha : \Pi(j : \underline{n}).\underline{m} + \underline{2}).\text{left-inj}(\alpha) \\
& \equiv \square_{\text{BCH}}^{\text{op}}(m, n)
\end{aligned}$$

Step 1 holds by Lemma 12. Let us look at Step 2. Giving a graph homomorphism between B_m and C_n corresponds to choosing where the origin is mapped to, and choosing where each (non-trivial) edge of B_m is mapped to. For the origin, we use the component $z : \underline{2}^{\underline{2}}$. There are m non-trivial edges in B_m , and z is an endpoint (or starting point) of n non-trivial edges and one trivial edge in C_n . This gives us up to $\underline{m} \rightarrow \underline{n} + \underline{1}$ possible functions, but since we only consider meet-preserving morphisms, every function needs to be injective on the left part, leading to $d : \underline{m} \xrightarrow{\text{left}} \underline{n} + \underline{1}$. Moreover, if $d(i) = \text{inl}(j)$ for some i, j , then the image of the origin must be the *starting point* of the edge in dimension j , i.e. $z(j) = 0$. Step 3 is an application of Lemma 2 (it essentially swaps the roles of m and n). Step 4 only unfolds the definition of $\xrightarrow{\text{left}}$.

In Step 5, the usual distributivity between Σ and Π (under the propositions-as-types view referred to as the “axiom of choice”) is used: z , e , and the unnamed last component can all be seen as (dependent) functions with domain \underline{n} . The dependent function α combines them into a single dependent function with domain \underline{n} and a codomain that consists of multiple components which, again, are called e , z , with the last one being unnamed. Only the component expressing the “injectivity on the left part”-property cannot be seen as a function in \underline{n} . In Step 6, we massage the codomain of α : We have $e : \underline{m} + \underline{1}$ and also $z : \underline{2}$, but the condition says that z is determined unless $e = \text{inr}(0)$; thus, the type is equivalent to $\underline{m} + \underline{2}$.

We omit the calculation which shows that the constructed equivalence preserves composition of morphisms in the categories. ◀

5:10 From Cubes to Twisted Cubes via Graph Morphisms in Type Theory

In Section 3, we will switch from standard cubes to twisted cubes. The directions of some edges will be reversed. It is therefore an advantage to formulate a condition similar to the one about meets and joins without referring to the direction of edges. This is indeed possible:

► **Definition 13** (dimension preserving morphisms; category \square_{dim}). *Given the standard cube C_n , where we use the non-recursive definition as in Definition 6, the dimension of an edge is defined as follows:*

$$\dim : \text{edges}(C_n) \rightarrow \underline{n} + \underline{1} \qquad \dim(\text{inl}(v)) \qquad \equiv \text{inr}(0) \qquad (24)$$

$$\dim(\text{inr}(i, x_0 \dots x_{n-2})) \equiv \text{inl}(i) \qquad (25)$$

We say that a morphism $f : \text{grp-hom}(C_m, C_n)$ is dimension-preserving if f maps edges of the same dimension to edges of the same dimension,

$$\text{dim-pres}(f) := \prod (e_1, e_2 : \text{edges}(C_n)). (\dim(e_1) = \dim(e_2)) \rightarrow (\dim(f(e_1)) = \dim(f(e_2))). \quad (26)$$

The category \square_{dim} makes use of these concepts:

$$\text{obj}(\square_{\text{dim}}) := \mathbb{N} \qquad \square_{\text{dim}}(m, n) := \Sigma (g : \text{grp-hom}(C_m, C_n)). \text{dim-pres}(g) \qquad (27)$$

As $\text{pres-meet}(g)$ and $\text{pres-join}(g)$, preserving the dimension as in (26) is a proposition in the sense of homotopy type theory (has at most one proof).

► **Remark 14.** For a graph morphism f as in the definition above, the following condition says that f is “injective on dimensions” (on the non-trivial part):

$$\begin{aligned} \text{dim-inj}(f) &:= \prod (e_1, e_2 : \text{edges}(C_m), j : \underline{n}). (\dim(f(e_1)) = \text{inl}(j) \times \dim(f(e_2)) = \text{inl}(j)) \\ &\rightarrow (\dim(e_1) = \dim(e_2)). \end{aligned}$$

However, note that this follows directly from $\text{dim-pres}(f)$: Assume e_1, e_2 are edges such that $\dim(f(e_1))$ and $\dim(f(e_2))$ are equal and non-trivial. If e_1 and e_2 are not “parallel” (i.e. not in the same dimension), then we can find e'_1 in the same dimension as e_1 such that e'_1 and e_2 are adjacent (i.e. the endpoint of one is the starting point of the other). It is clear that $f(e'_1)$ and $f(e_2)$ cannot go into the same non-trivial direction, since we can only go one step into a given direction before going back.

The connection to meet- and join-preserving is given by the following result:

► **Lemma 15.** *A morphism $f : \text{grp-hom}(C_m, C_n)$ is join-and-meet-preserving exactly if it is dimension-preserving.*

Proof. This follows easily by going via morphisms $\text{grp-hom}(B_m, C_n)$ as in Lemma 12. The graph B_m has exactly one edge for every non-trivial dimension, and the proof is analogous to the one of Lemma 12. ◀

► **Corollary 16** (Section summary). *The categories $\square_{\text{BCH}}^{\text{op}}$, \square_{cont} , and \square_{dim} are isomorphic.* ◀

3 A Category of Twisted Cubes

As discussed in the introduction, we build on our framework of graph morphisms to define a category of *twisted cubes*. A variation of Definition 4 gives us these twisted cubes. The critical change can be seen in (29), which should be compared with (6):

► **Definition 17.** Given a graph $G = (V, E)$, the twisted graph-prism of G , denoted as $\text{tw-prism}(G) := (\text{tw-prism}(V), \text{tw-prism}(E))$ is the graph defined by

$$\text{tw-prism}(V) := \underline{2} \times V \tag{28}$$

$$\text{tw-prism}(E) := \{ ((0, t), (0, s)) \mid (s, t) : E \} \tag{29}$$

$$\cup \{ ((1, s), (1, t)) \mid (s, t) : E \} \tag{30}$$

$$\cup \{ ((0, v), (1, v)) \mid v : V \}. \tag{31}$$

We then define:

► **Definition 18.** Given $n : \mathbb{N}$, the twisted cube T_n is defined as follows:

$$T_0 := (\underline{1}, \{(0,0)\}) \qquad T_{n+1} := \text{tw-prism}(T_n) \tag{32}$$

Alternatively, we can tweak Definition 5 to get a non-recursive definition. As before, the convention is that $\underline{-1}$ is empty.

► **Definition 19.** The non-recursive definition of T_n is as follows:

$$\text{nodes}(T_n) \qquad \qquad \qquad \equiv \qquad \underline{2}^n \tag{33}$$

$$\text{edges}(T_n) \qquad \qquad \qquad \equiv \qquad \underline{2}^n + (\underline{n} \times \underline{2}^{n-1}) \tag{34}$$

$$\text{src}(\text{inl}(v)) \quad \equiv \quad \text{trg}(\text{inl}(v)) \quad \equiv \quad v \tag{35}$$

$$\text{src}(\text{inr}(i, x_0 x_1 \dots x_{n-2})) \quad \equiv \quad x_0 x_1 \dots x_{i-1} \cdot b \cdot x_i \dots x_{n-2} \tag{36}$$

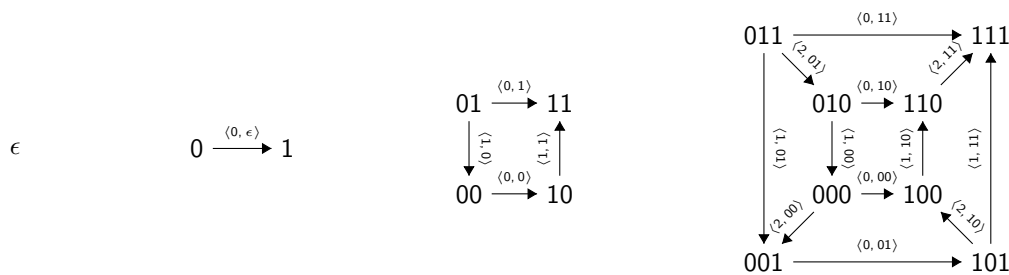
$$\text{trg}(\text{inr}(i, x_0 x_1 \dots x_{n-2})) \quad \equiv \quad x_0 x_1 \dots x_{i-1} \cdot (1 - b) \cdot x_i \dots x_{n-2} \tag{37}$$

where $b = 1$ if the total number of zeros in $x_0 x_1 \dots x_{i-1}$ is odd, and $b = 0$ otherwise.

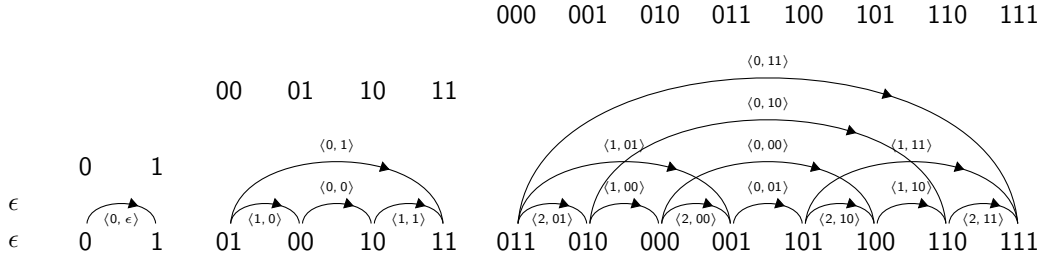
This means that an edge is reversed (compared to the standard cubes discussed before) exactly if the number of zeros in dimensions that come *before* the edge is odd (note that the condition talks about x_{i-1} , not x_{n-2}). The twisted cubes of dimension up to 3 are illustrated in Figure 6; see also Figures 3 and 4 in the introduction.

► **Lemma 20.** Definition 18 and Definition 19 define isomorphic graph structures. ◀

T_n has an interesting property that the standard cube C_n does not have: The induced preorder T_n^* on the vertices is a total order. This observation was originally suggested by Paolo Capriotti and Jakob von Raumer in a discussion with the first author of this paper. Note that this observation should not be misunderstood to mean that T_n itself is uninteresting. Its edges give it a unique structure, as visualised in Figure 7.



■ **Figure 6** An illustration of T_n where $n \leq 3$.



■ **Figure 7** Linear drawings of the twisted cubes T_0 , T_1 , T_2 , and T_3 , demonstrating that the underlying preorders are total orders. The binary sequences on top are the values of g_n from the proof of Theorem 21. See also Remark 22.

The idea behind this result is that *tw-prism* preserves the property of having a preorder that is total. To elaborate on this, if G^* is a total order, then $(\text{tw-prism } G)^*$ consists of two copies of G^* , where the first copy is “turned around”. One of the edges added in (31) links the largest node in the first copy to the smallest node in the second copy, thus every element of the second copy is larger than all the elements of the first copy. In other words, $(\text{tw-prism } G)^*$ is the *join* of the two copies.⁵

► **Theorem 21.** *For all $n : \mathbb{N}$, the preorder T_n^* is isomorphic to the total order $(\underline{2}^n, <)$.*

Note that Theorem 21 is a property which one usually expects for simplicial structures, but not for cubical ones.

► **Remark 22.** There are two binary numbers for each node in Figure 7. The bottom one represents each node name according to Definition 19 whereas the top one represents the total order of T_3 . It is impossible to unify these two binary numbers for $n \geq 2$ since, for each edge e , the numbers $\text{src}(e)$ and $\text{tgt}(e)$ only differ by (at most) one single bit by Definition 19, while incrementing a binary number can flip more than one bit.

Another related observation is that we can find a path from the smallest vertex to the largest vertex of T_n which respects the direction of the edges, and which visits each vertex exactly once. Recall that such a path is called a *Hamiltonian path*. We record this:

► **Theorem 23.** *For all $n : \mathbb{N}$, there is exactly one Hamiltonian path through T_{n+1} . This path contains exactly one edge in the first dimension (i.e. the one which is added when going from T_n to T_{n+1}). Moreover, this single edge in the new dimension connects the Hamiltonian paths through the two copies of T_n of which T_{n+1} consists by definition, cf. (28).*

Proof of Theorem 21 and Theorem 23. As before, we denote elements of $\underline{2}^n$ as sequences such as 00101 (binary representation with most significant bit first) or, for clarity, by 0·0·1·0·1. We use the endofunction **neg** on $\underline{2}^n$, which simply replaces each 0 in a sequence by a 1 and vice versa; i.e. it sends the number i to $2^n - 1 - i$ (note that **neg** does not reverse the sequence, but the ordering on $\underline{2}^n$).

⁵ *Join* in the sense of the *join of categories* [19], which should not be confused with the join (coproduct) of objects in a preorder (cf. Definition 9).

Let us define endofunctions f_n and g_n on $\underline{2}^n$, by induction on n . Note that, at this point, we do not talk about graph morphisms but only about functions between sets. The base cases of the induction are uniquely determined. We define f and g by

$$f_{n+1}(0 \cdot \vec{x}) := 0 \cdot f_n(\text{neg}(\vec{x})) \qquad g_{n+1}(0 \cdot \vec{x}) := 0 \cdot \text{neg}(g_n(\vec{x})) \qquad (38)$$

$$f_{n+1}(1 \cdot \vec{x}) := 1 \cdot f_n(\vec{x}) \qquad g_{n+1}(1 \cdot \vec{x}) := 1 \cdot g_n(\vec{x}). \qquad (39)$$

It is easy to calculate that, by induction, f and g are inverse to each other. We want to show that they extend to morphisms between preorders,

$$\widehat{f}_n : (\underline{2}^n, <) \rightarrow T_n^* \qquad \widehat{g}_n : T_n^* \rightarrow (\underline{2}^n, <). \qquad (40)$$

To construct \widehat{f}_n and the Hamiltonian path through the cube, it suffices to show: for $x, y : \underline{2}^n$ with $x + 1 = y$, we have an edge $f_n(x) \rightarrow f_n(y)$.

We do induction on n . For $n = 0$, this is vacuously true (such x, y do not exist). For $n = n' + 1$, there are multiple cases:

- case $x = 0 \cdot x'$ and $y = 0 \cdot y'$: Then, the assumption gives us $x' + 1 = y'$ and we have to find an edge $0 \cdot f_n(\text{neg}(x')) \rightarrow 0 \cdot f_n(\text{neg}(y'))$. Looking at Definition 17, we can get this if we have $f_n(\text{neg}(y')) \rightarrow f_n(\text{neg}(x'))$. This holds by induction, since neg reverses the order which gives us $\text{neg}(y') + 1 = \text{neg}(x')$.
- case $x = 1 \cdot x'$ and $y = 1 \cdot y'$: Similar to the previous case, but nothing gets reversed.
- case $x = 0 \cdot x'$ and $y = 1 \cdot y'$: In this case, we have $x = 0111 \dots$ and $y = 1000 \dots$. We need to find an edge $0 \cdot f(\text{neg}(111 \dots)) \rightarrow 1 \cdot f(000 \dots)$, which simplifies to $0 \cdot f(000 \dots) \rightarrow 1 \cdot f(000 \dots)$. This edge is directly given in (31).
- case $x = 1 \cdot x'$ and $y = 0 \cdot y'$: Contradicts with the assumption $x + 1 = y$.

This shows that there is a Hamiltonian path, and it is given by \widehat{f}_n . The definition of f as in (38,39) also shows that f_{n+1} consists of two copies of f_n , implying the last claim of Theorem 23. In order to prove Theorem 21, we need to construct \widehat{g}_n . It is enough to show that, for an edge from u to v in T_n , we have $g(u) \leq g(v)$. This follows by straightforward induction, going through the edges in Definition 17. But Theorem 21 implies that there is at most one Hamiltonian path. ◀

► **Remark 24.** Note that every vertex v in T_n is an endpoint of n non-trivial edges. The number of zeros in the binary representation in the “order number” of v (i.e. the value $g_n(v)$ in the proof of Theorem 21) equals the number of *outgoing* edges. Figure 7 shows this.

Analogously to Definition 8, we can now define the category of twisted graph morphisms:

► **Definition 25** (category $\mathfrak{N}_{\text{grp}}$). *The category $\mathfrak{N}_{\text{grp}}$ has natural numbers as objects, and morphisms from m to n are graph morphisms between twisted cubes:*

$$\text{obj}(\mathfrak{N}_{\text{grp}}) := \mathbb{N} \qquad \mathfrak{N}_{\text{grp}}(m, n) := \text{grp-hom}(T_m, T_n) \qquad (41)$$

It is easy to see that the category $\mathfrak{N}_{\text{grp}}$ has a version of connections. Since we are looking for a “twisted analogue” of $\square_{\text{BCH}}^{\text{op}}$, we need to refine it further. In Section 2, we have discussed the restriction to (meet and join)-preserving morphisms, and to dimension-preserving morphisms. It follows directly from Theorem 21 that every morphism in $\mathfrak{N}_{\text{grp}}$ preserves all binary meets and joins, so this condition becomes trivial; it does not avoid connections. However, preserving dimensions is still a non-trivial condition which does avoid connections. The definition of equation (26) still works.

► **Definition 26** (category \mathfrak{A}_{\dim}). *The category \mathfrak{A}_{\dim} has dimension-preserving maps between twisted cubes as morphisms:*

$$\text{obj}(\mathfrak{A}_{\dim}) := \mathbb{N} \qquad \mathfrak{A}_{\dim}(m, n) := \Sigma(g : \text{grp-hom}(T_m, T_n)).\text{dim-pres}(g) \quad (42)$$

Note that the explanation of Remark 14 holds for the twisted cube category as well.

A consequence of Theorem 21 is that morphisms in \mathfrak{A}_{\dim} cannot “swap dimensions”. But an even stronger result holds, namely that surjective morphisms are unique:

► **Theorem 27.** *There is exactly one surjective morphism in $\mathfrak{A}_{\dim}(m, n)$ for $m \geq n$. (Clearly, there is none if $m < n$.)*

Proof. The key to the proof is Theorem 23. Clearly, the Hamiltonian path in T_m goes through all vertices. Due to surjectivity, its image has to go through all vertices of T_n . In other words, the T_m -Hamiltonian path has to be mapped to the T_n -Hamiltonian path. Since the graph morphisms that we consider preserve the dimension, the only edge in the T_m -path which can be mapped to the single edge in the first dimension in the T_n -path is just this single edge in the first dimension in the T_m -path; i.e. the middle edge has to be mapped to the middle edge. From here, it follows by induction that there can only be at most one surjective graph morphism.

What is left to show is that there actually is a surjective graph morphism if $m \geq n$. It is enough to construct a surjective graph morphism $f : \mathfrak{A}_{\dim}(n+1, n)$, from where we get any other by $(m-n)$ -fold composition (0-fold composition is the identity). Such a graph morphism is given by

$$f(x_0 \dots x_{n-1} x_n) := (x_0 \dots x_{n-1}). \quad (43)$$

Since the directions of the edges do not depend on the very last dimension, this works (cf. Definition 19). ◀

An important consequence of the above result is that there is a unique way to degenerate a twisted cube. We do not go into the details here, but see the conclusions at the end of the paper. For now, we go into a different direction.

Let us write intv (“interval”) for the finite set $\{0, 1, \star\}$. Of course, intv is isomorphic to $\underline{3}$, but referring to the last element as \star helps the intuition, we hope.

► **Definition 28.** *A face of the twisted n -cube T_n is a function $f : \underline{n} \rightarrow \text{intv}$. The dimension of a face, written $\text{dim}(f)$, equals the number of times f takes \star as value (i.e. the size of $f^{-1}(\star)$). The type of faces of dimension k is written as $\text{faces}(n, k)$.*

The face $f : \underline{n} \rightarrow \text{intv}$ represents the full subgraph of T_n of vertices on which f “matches” (a vertex $x_0 x_1 \dots x_{n-1}$ is matched if, for every i , we have $f(i) = x_i$ or $f(i) = \star$).

► **Lemma 29.** *The image of $f : \mathfrak{A}_{\dim}(m, n)$ is a face.*

Proof. This follows from the property of preserving the dimension as defined in (26). ◀

► **Lemma 30.** *The m -faces are the only injective maps $\mathfrak{A}_{\dim}(m, n)$:*

$$\text{faces}(n, m) \simeq \Sigma(f : \mathfrak{A}_{\dim}(m, n)).\text{is-inj}(f). \quad (44)$$

Proof. Every face gives rise to a canonical injective dimension-preserving morphism in the sense of Definition 13, as dictated by the inclusion of the full subgraph that the face represents into T_n . The fact that these are the only ones follows from Theorem 21 (we cannot “swap dimensions”) and Lemma 29. ◀

As with Theorem 21 before, Lemma 30 is a result which is usually found in simplicial structures, but not in cubical ones. In any case, we now easily get:

► **Lemma 31** (factorisation of dimension preserving morphisms). *Given a morphism $f : \mathfrak{A}_{\dim}(m, n)$, there is exactly one way to write it as the composition $f = \text{inj}(f) \circ \text{surj}(f)$ of a surjective dimension preserving graph morphism followed by an injective one. This means that the map*

$$(\Sigma(k : \mathbb{N}). (\Sigma(h : \mathfrak{A}_{\dim}(k, n)). \text{is-inj}(h)) \times (\Sigma(g : \mathfrak{A}_{\dim}(m, k)). \text{is-surj}(g))) \rightarrow \mathfrak{A}_{\dim}(m, n) \quad (45)$$

$$(k, (h, i), (g, s)) \mapsto h \circ g \quad (46)$$

is an equivalence. Moreover, morphisms $\mathfrak{A}_{\dim}(m, n)$ are in 1-to-1 correspondence with faces of T_n of dimension $\leq m$.

Proof. A consequence of Lemma 29 is that the factorisation on the level of sets of vertices works. The second claim follows from the first: In (45), the k and the surjective map are uniquely determined (i.e. contractible components) by Theorem 27. By Lemma 30, injective maps correspond to faces. ◀

► **Remark 32.** It follows from Lemma 31 and the proof of Theorem 27 that all the non-empty fibres of a dimension-preserving morphism between twisted cubes have the same size. The reverse is the case as well: a morphism between twisted graphs where all non-empty fibres have the same size is dimension-preserving.

Another consequence of the above results is that \mathfrak{A}_{\dim} can be given the structure of a *Reedy category* (cf. [14]). Recall that a Reedy category is a category R with a degree function $d : \text{obj}(\mathfrak{A}_{\dim}) \rightarrow \mathbb{N}$ and two subcategories R^+ and R^- , such that:⁶

- both subcategories are *wide*, i.e. contain all the objects of R ;
- every nonidentity morphism in R^+ raises the degree;
- every nonidentity morphism in R^- lowers the degree;
- and every morphism of R can be written as a morphism in R^- followed by a morphism in R^+ in a unique way.

The reason why Reedy categories are interesting is that they enable certain inductive constructions. In the setting of type theory, they have been discussed by Shulman [26].

► **Theorem 33.** *The category \mathfrak{A}_{\dim} is a Reedy category where the degree of an object is the object itself (recall that objects are natural numbers). \mathfrak{A}_{\dim}^+ is the subcategory of injective morphisms, and \mathfrak{A}_{\dim}^- is the subcategory of surjective morphisms.*

Proof. The first three properties are clear, and the factorisation is given by Lemma 31. ◀

Finally, let us record an alternative representation of the category \mathfrak{A}_{\dim} which does not go via graph morphisms.

► **Definition 34** (ternary notation: category $\mathfrak{A}_{\text{tri}}$). *The category $\mathfrak{A}_{\text{tri}}$ has natural numbers as objects, and a morphism from m to n is a function $\underline{n} \rightarrow \text{intv}$ which takes \star at most m times as image:*

$$\text{obj}(\mathfrak{A}_{\text{tri}}) := \mathbb{N} \qquad \mathfrak{A}_{\text{tri}}(m, n) := \Sigma(f : \underline{n} \rightarrow \text{intv}). f^{-1}(\star) \leq m \quad (47)$$

⁶ Degrees can more generally be arbitrary ordinals, but \mathbb{N} is sufficient in our case.

The identity morphisms are the functions that are constantly \star . To define the composition of $f : \mathfrak{N}_{\text{tri}}(k, m)$ and $g : \mathfrak{N}_{\text{tri}}(m, n)$, we need to define a function $g \circ f : \underline{n} \rightarrow \text{intv}$ (which is \star at most k times). We define $(g \circ f)(i)$ by recursion on i , simultaneously with the values i' and b_i , as follows:

$$(g \circ f)(i) := \begin{cases} g(i) & \text{if } g(i) \in \{0, 1\} \\ (f(i')) \text{ xor } b_i & \text{if } g(i) = \star \text{ and } f(i') \in \{0, 1\} \\ \star & \text{if } g(i) = \star \text{ and } f(i') = \star \end{cases} \quad (48)$$

where

- i' is the number of occurrences of \star in the sequence $g(0), g(1), \dots, g(i-1)$;
- b_i is 1 if the number of zeros in the sequence $(g \circ f)(0), (g \circ f)(1), \dots, (g \circ f)(i-1)$ is odd, and 0 if it is even.

Note that a morphism in $\mathfrak{N}_{\text{tri}}(m, n)$ can be represented as a sequence such as $01\star 0\star 10$ of length n which contains the symbol \star at most m times, which is why we refer to it as *ternary notation*.

► **Remark 35.** There is a category of twisted semi-cubes, denoted by $\mathfrak{N}_{\text{tri}}^+$, which is exactly the same as $\mathfrak{N}_{\text{tri}}$ except that the number of \star in the sequence must be exactly m , i.e. “ \leq ” is changed to “ $=$ ” in the definition of $\mathfrak{N}_{\text{tri}}(m, n)$. This category is equivalent to the subcategory of $\mathfrak{N}_{\text{dim}}$, denoted as $\mathfrak{N}_{\text{dim}}^+$, which consists of *injective* dimension-preserving graph homomorphisms. Note that this injectivity condition is equivalent to removing the reflexive edges from Definition 18.

If we remove the expression $(\text{xor } b_i)$ in the definition of morphisms of $\mathfrak{N}_{\text{tri}}^+$, then the category becomes equivalent to the category of standard cubes but without degeneracies and swapping dimensions. In other words, the expression $(\text{xor } b_i)$ characterises “twisted-ness”.

► **Theorem 36.** *The categories $\mathfrak{N}_{\text{dim}}$, and $\mathfrak{N}_{\text{tri}}$ are isomorphic, with the object part being the identity. In particular, we have:*

$$\mathfrak{N}_{\text{dim}}(m, n) \simeq \mathfrak{N}_{\text{tri}}(m, n) \quad (49)$$

Proof. As the following chain of equivalences:

$$\begin{aligned} & \mathfrak{N}_{\text{dim}}(m, n) \\ \text{[Lemma 31]} & \simeq \Sigma(k : \mathbb{N}). (\Sigma(h : \mathfrak{N}_{\text{dim}}(k, n)). \text{is-inj}(h)) \times (\Sigma(g : \mathfrak{N}_{\text{dim}}(m, k)). \text{is-surj}(g)) \\ \text{[Theorem 27]} & \simeq \Sigma(k : \mathbb{N}). (\Sigma(h : \mathfrak{N}_{\text{dim}}(k, n)). \text{is-inj}(h)) \times (k \leq m) \\ \text{[Lemma 30]} & \simeq \Sigma(k : \mathbb{N}). \text{faces}(n, k) \times (k \leq m) \\ \text{[simplification]} & \simeq \Sigma(f : \underline{n} \rightarrow \text{intv}). f^{-1}(\star) \leq m \\ & \equiv \mathfrak{N}_{\text{tri}}(m, n) \end{aligned}$$

When transported along this isomorphism, the composition of $\mathfrak{N}_{\text{dim}}$ gets mapped to the composition of $\mathfrak{N}_{\text{tri}}$, as required. ◀

4 Conclusions and Future Directions

We have suggested new representations of the BCH cube category and introduced a category of twisted cubes. It is natural to further study the similarities and differences between standard and twisted cube categories, and some new results will be presented in the upcoming PhD thesis of the first author.

As future work, we plan to examine algebraic descriptions via generators and relations. Such presentations exist for many different cube categories in the literature but, as far as we are aware, not for the BCH cube category. The closest suggestions available are the presentations by Antolini [5] and Newstead [20], which seem to be fairly easy to adapt to the BCH cube category. Interestingly, further adapting the generators to the *twisted* setting simplifies them significantly, which mirrors the fact that morphisms between twisted cubes cannot swap dimensions. Moreover, our Theorem 27 implies that degeneracies are unique: there is only one single way in which a twisted n -cube can be degenerated to get a twisted $(n + 1)$ -cube. A consequence is that we do not need to impose relations between different degeneracies.

This, we hope, will make it possible to develop the higher categorical structures that can be encoded as presheaves on the category of twisted cubes. An ultimate goal would be to model some form of *directed cubical type theory* mirroring the model by Bezem, Coquand, and Huber [8].

Another possible application of our twisted cube categories might be building a syntax for a parametric type theory or cubical type theory without an interval as suggested by Altenkirch and Kaposi [2]. A major difficulty in their development was the presence of multiple degeneracies, a problem which does not occur in the current work.

A further direction which may be worth exploring is to not consider set-valued presheaves, but type-valued presheaves instead. To facilitate this, we can consider the category of twisted semi-cubes mentioned in Remark 35. From there, type-valued presheaves can be encoded as Reedy-fibrant diagrams in a known style [27]. We can then add a condition reminiscent of Rezk's *Segal-condition* [24] by stating that the projection from twisted semi-cubical types to the sequence of types along the Hamiltonian path is an equivalence. This corresponds to saying that the partial n -cube with missing inner part and lid (cf. Figure 3) have a contractible type of fillers. It seems that this could be a first step towards the construction of composition and higher coherences, although further conditions seem to be necessary. The relation to the (*complete*) *semi-Segal types* by Capriotti and others [4, 10, 11] remains to be studied.

References

- 1 I. R. Aitchison. The geometry of oriented cubes. *ArXiv e-prints*, 2010. [arXiv:1008.1714](https://arxiv.org/abs/1008.1714).
- 2 Thorsten Altenkirch and Ambrus Kaposi. Towards a Cubical Type Theory without an Interval. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:27, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2015.3.
- 3 Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. Cartesian cubical type theory. URL: <https://github.com/dlicata335/cart-cube/blob/master/cart-cube.pdf>.
- 4 Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *ArXiv e-prints*, 2019. [arXiv:1705.03307](https://arxiv.org/abs/1705.03307).
- 5 Rosa Antolini. Geometric realisations of cubical sets with connections, and classifying spaces of categories. *Applied Categorical Structures*, 2002.
- 6 Steve Awodey. A cubical model of homotopy type theory. *Annals of Pure and Applied Logic*, 2018.
- 7 Hans-Joachim Baues and Günther Wirsching. Cohomology of small categories. *Journal of pure and applied algebra*, 38(2-3):187–211, 1985. See also <https://ncatlab.org/nlab/show/category+of+factorizations>.
- 8 Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, 2014.

- 9 Ulrik Buchholtz and Edward Morehouse. Varieties of cubical sets. *Relational and Algebraic Methods in Computer Science*, 2017.
- 10 Paolo Capriotti. *Models of Type Theory with Strict Equality*. PhD thesis, School of Computer Science, University of Nottingham, Nottingham, UK, 2016. Available online at [arXiv:1702.04912](https://arxiv.org/abs/1702.04912).
- 11 Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-segal types. *Proceedings of the ACM on Programming Languages*, 2(POPL'18):44:1–44:29, December 2017. Full version available at [arXiv:1707.03693](https://arxiv.org/abs/1707.03693). doi:10.1145/3158132.
- 12 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2015.5.
- 13 Nicola Gambino and Christian Sattler. The frobenius condition, right properness, and uniform fibrations. *Journal of Pure and Applied Algebra*, 221(12):3027–3068, 2017.
- 14 Philip S Hirschhorn. *Model categories and their localizations*. American Mathematical Soc., 2009.
- 15 Simon Huber. *Cubical Interpretations of Type Theory*. PhD thesis, Department of Computer Science and Engineering, University of Gothenburg, 2016.
- 16 Chris Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after voevodsky). *ArXiv e-prints*, November 2012. To appear in the Journal of the European Mathematical Society. [arXiv:1211.2851](https://arxiv.org/abs/1211.2851).
- 17 F William Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. *Applications of Categorical Algebra*, 17:1–14, 1970. See also <https://ncatlab.org/nlab/show/twisted+arrow+category>.
- 18 Daniel R Licata and Robert Harper. 2-dimensional directed type theory. *Electronic Notes in Theoretical Computer Science*, 276:263–289, 2011.
- 19 Jacob Lurie. *Higher Topos Theory*. Annals of Mathematics Studies. Princeton University Press, Princeton, 2009. Also available online at [arXiv:math/0608040](https://arxiv.org/abs/math/0608040); see also <https://ncatlab.org/nlab/show/join+of+categories>.
- 20 Clive Newstead. Cubical sets. URL: math.cmu.edu/~cnewstea/notes/cubicalsets.pdf.
- 21 Paige Randall North. Towards a directed homotopy type theory. *Electronic Notes in Theoretical Computer Science*, 347:223–239, 2019.
- 22 Andreas Nuyts. Towards a directed homotopy type theory based on 4 kinds of variance. Master's thesis, KU Leuven, 2015.
- 23 I. Orton and A. M. Pitts. Axioms for modelling cubical type theory in a topos. *Logical Methods in Computer Science*, 2018. Special issue for CSL 2016.
- 24 Charles Rezk. A model for the homotopy theory of homotopy theory. *Transactions of the American Mathematical Society*, 2001.
- 25 Emily Riehl and Michael Shulman. A type theory for synthetic ∞ -categories. *Higher Structures*, 1(1), 2017. URL: https://journals.mq.edu.au/index.php/higher_structures/article/view/36.
- 26 Michael Shulman. The univalence axiom for elegant Reedy presheaves. *Homology, Homotopy and Applications*, 2015. doi:10.4310/HHA.2015.v17.n2.a6.
- 27 Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, 2015.
- 28 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- 29 Vladimir Voevodsky. Univalent foundations project. A modified version of an NSF grant application, 2010.


For Finitary Induction-Induction, Induction Is Enough

Ambrus Kaposi 

Eötvös Loránd University, Budapest, Hungary
akaposi@inf.elte.hu

András Kovács 

Eötvös Loránd University, Budapest, Hungary
kovacsandras@inf.elte.hu

Ambroise Lafont 

IMT Atlantique, Inria, LS2N CNRS, Nantes, France
ambroise.lafont@gmail.com

Abstract

Inductive-inductive types (IITs) are a generalisation of inductive types in type theory. They allow the mutual definition of types with multiple sorts where later sorts can be indexed by previous ones. An example is the Chapman-style syntax of type theory with conversion relations for each sort where e.g. the sort of types is indexed by contexts. In this paper we show that if a model of extensional type theory (ETT) supports indexed W -types, then it supports finitely branching IITs. We use a small internal type theory called the theory of signatures to specify IITs. We show that if a model of ETT supports the syntax for the theory of signatures, then it supports all IITs. We construct this syntax from indexed W -types using preterms and typing relations and prove its initiality following Streicher. The construction of the syntax and its initiality proof were formalised in Agda.

2012 ACM Subject Classification Theory of computation \rightarrow Type theory

Keywords and phrases type theory, inductive types, inductive-inductive types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.6

Supplementary Material The contents of Section 4 were formalised in Agda, the formalisation is available at <https://github.com/ambroise/UniversalIIT>.

Funding *Ambrus Kaposi*: this author was supported by the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme, Project no. ED_18-1-2019-0030, by the New National Excellence Program of the Ministry for Innovation and Technology, Project no. ÚNKP-19-4-ELTE-874, and by the Bolyai Fellowship of the Hungarian Academy of Sciences, Project no. BO/00659/19/3.

András Kovács: this author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Ambroise Lafont: this author was supported by the CoqHoTT ERC Grant 637339.

Acknowledgements The authors would like to thank Thorsten Altenkirch, Rafaël Bocquet, Simon Boulier, Fredrik Nordvall-Forsberg and Jakob von Raumer for discussions on the topics of this paper. We also thank the anonymous reviewers for their helpful comments and suggestions.

1 Introduction

Many mutual inductive types can be reduced to indexed inductive types, where the index disambiguates different sorts. For example, consider the mutual inductive datatype with two sorts `isEven` and `isOdd`, defined by the following constructors.

`isEven` : $\mathbb{N} \rightarrow \text{Set}$

`isOdd` : $\mathbb{N} \rightarrow \text{Set}$

`zeroEven` : `isEven zero`



© Ambrus Kaposi, András Kovács, and Ambroise Lafont;
licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 6; pp. 6:1–6:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

6:2 For Finitary Induction-Induction, Induction Is Enough

$$\begin{aligned} \text{sucEven} & : (n : \mathbb{N}) \rightarrow \text{isOdd } n \rightarrow \text{isEven } (\text{suc } n) \\ \text{sucOdd} & : (n : \mathbb{N}) \rightarrow \text{isEven } n \rightarrow \text{isOdd } (\text{suc } n) \end{aligned}$$

This can be reduced to the following single inductive family where `isEven? true` represents `isEven` and `isEven? false` represent `isOdd`.

$$\begin{aligned} \text{isEven?} & : \text{Bool} \rightarrow \mathbb{N} \rightarrow \text{Set} \\ \text{zeroEven} & : \text{isEven? true zero} \\ \text{sucEven} & : (n : \mathbb{N}) \rightarrow \text{isEven? false } n \rightarrow \text{isEven? true } (\text{suc } n) \\ \text{sucOdd} & : (n : \mathbb{N}) \rightarrow \text{isEven? true } n \rightarrow \text{isEven? false } (\text{suc } n) \end{aligned}$$

Inductive-inductive types (IITs [26]) allow the mutual definition of a type and a family of types over the first one. IITs were originally introduced to represent the well-typed syntax of type theory itself, and a prominent example is still Chapman’s [13] syntax for a type theory. A minimised version is the IIT of contexts and types given by the following constructors.

$$\begin{aligned} \text{Con} & : \text{Set} \\ \text{Ty} & : \text{Con} \rightarrow \text{Set} \\ \text{empty} & : \text{Con} \\ \text{ext} & : (\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Con} \\ \text{U} & : (\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \\ \text{El} & : (\Gamma : \text{Con}) \rightarrow \text{Ty } (\text{ext } \Gamma (\text{U } \Gamma)) \end{aligned}$$

This type has two sorts, `Con` and `Ty`. The `ext` constructor of `Con` refers to `Ty` and the `Ty`-constructor `U` refers to `Con`, hence the two sorts have to be defined simultaneously. Moreover, `Ty` is indexed over `Con`. This precludes a reduction analogous to the reduction of `isEven–isOdd`, as we would get a type indexed over itself. Another unique feature of IITs (which also holds for higher inductive types [29]) is that later constructors can refer to previous constructors: in our case, `El` mentions `ext`.

The elimination principle for the above IIT has the following two motives (one for each sort) and four methods (one for each constructor).

$$\begin{aligned} \text{Con}^D & : \text{Con} \rightarrow \text{Set} \\ \text{Ty}^D & : \text{Con}^D \Gamma \rightarrow \text{Ty } \Gamma \rightarrow \text{Set} \\ \text{empty}^D & : \text{Con}^D \text{ empty} \\ \text{ext}^D & : (\Gamma^D : \text{Con}^D \Gamma) \rightarrow \text{Ty}^D \Gamma^D A \rightarrow \text{Con}^D (\text{ext } \Gamma A) \\ \text{U}^D & : (\Gamma^D : \text{Con}^D \Gamma) \rightarrow \text{Ty}^D \Gamma^D (\text{U } \Gamma) \\ \text{El}^D & : (\Gamma^D : \text{Con}^D \Gamma) \rightarrow \text{Ty}^D (\text{ext}^D \Gamma^D (\text{U}^D \Gamma^D)) (\text{El } \Gamma) \end{aligned}$$

Above we used implicit quantifications for $\Gamma : \text{Con}$ and $A : \text{Ty } \Gamma$ to ease readability, e.g. Ty^D has an implicit parameter Γ before its explicit parameter of type $\text{Con}^D \Gamma$.

Given the above motives and methods the elimination principle provides two functions

$$\begin{aligned} \text{elimCon} & : (\Gamma : \text{Con}) \rightarrow \text{Con}^D \Gamma \\ \text{elimTy} & : (A : \text{Ty } \Gamma) \rightarrow \text{Ty}^D (\text{elimCon } \Gamma) A \end{aligned}$$

with the following computation rules.

$$\begin{aligned}
\text{elimCon empty} &= \text{empty}^D \\
\text{elimCon (ext } \Gamma A) &= \text{ext}^D (\text{elimCon } \Gamma) (\text{elimTy } A) \\
\text{elimTy (U } \Gamma) &= U^D (\text{elimCon } \Gamma) \\
\text{elimTy (El } \Gamma) &= El^D (\text{elimCon } \Gamma)
\end{aligned}$$

The functions `elimCon` and `elimTy` are an example of a *recursive-recursive* definition (using nomenclature from [26]). This means two mutually defined functions where the type of the second function depends on the first function. The proof assistant Agda [28] allows defining such functions (even from non-IITs) and is currently the only proof assistant supporting IITs¹.

Reducing IITs to inductive types (more precisely, to indexed W-types) is an open problem. Forsberg [26] presented a reduction in extensional type theory, however, this only provides a simpler, non-recursive-recursive elimination principle. Hugunin [19] reduced several IITs to inductive types, working inside a cubical type theory, but he also only constructed the simple eliminator. To illustrate the difference, we list the motives, methods and the simple elimination principle for the `Con`–`Ty` example. Again, we use implicit quantifications.

$$\begin{aligned}
\text{Con}^S &: \text{Con} \rightarrow \text{Set} \\
\text{Ty}^S &: \text{Ty } \Gamma \rightarrow \text{Set} \\
\text{empty}^S &: \text{Con}^S \text{ empty} \\
\text{ext}^S &: \text{Con}^S \Gamma \rightarrow \text{Ty}^S A \rightarrow \text{Con}^S (\text{ext } \Gamma A) \\
U^S &: \text{Con}^S \Gamma \rightarrow \text{Ty}^S (\text{U } \Gamma) \\
El^S &: \text{Con}^S \Gamma \rightarrow \text{Ty}^S (\text{El } \Gamma) \\
\text{selimCon} &: (\Gamma : \text{Con}) \rightarrow \text{Con}^S \Gamma \\
\text{selimTy} &: (A : \text{Ty } \Gamma) \rightarrow \text{Ty}^S A
\end{aligned}$$

This simple elimination principle is not capable of defining standard (metacircular) interpretation [4] of our small syntax. Using pattern matching notation, this interpretation is the following:

$$\begin{aligned}
\llbracket - \rrbracket &: \text{Con} \rightarrow \text{Set}_1 \\
\llbracket - \rrbracket &: \llbracket \Gamma \rrbracket \rightarrow \text{Set}_1 \\
\llbracket \text{empty} \rrbracket &:= \top \\
\llbracket \text{ext } \Gamma A \rrbracket &:= (\gamma : \llbracket \Gamma \rrbracket) \times \llbracket A \rrbracket \gamma \\
\llbracket \text{U } \Gamma \rrbracket \gamma &:= \text{Set} \\
\llbracket \text{El } \Gamma \rrbracket (\gamma, X) &:= X
\end{aligned}$$

The reason that we need the general elimination principle to define $\llbracket - \rrbracket$ is that $\llbracket - \rrbracket$ for types refers to $\llbracket - \rrbracket$ for contexts, hence this function is recursive-recursive.

Kaposi, Kovács, and Altenkirch [21] introduced a small type theory, called the theory of signatures, to describe quotient inductive-inductive types (QIIT). QIITs are generalisations of IITs where equality constructors are also allowed. A QIIT signature is a context in

¹ An experimental version of Coq with IITs is also available on GitHub.

the theory of QIIT signatures, for example natural numbers are specified by the context $(Nat : U, zero : Nat, suc : Nat \rightarrow Nat)$ of length three (Nat , $zero$ and suc are variable names). The theory of QIIT signatures is itself a QIIT. In *ibid.*, it is proved that if a model of extensional type theory supports the theory of QIIT signatures, then it supports all QIITs.

By omitting the equality type former from the theory of QIIT signatures, we obtain a theory of IIT signatures and the construction is still valid. It follows that if a model of extensional type theory supports the theory of IIT signatures, it supports all IITs.

In this paper we show that any model of extensional type theory with indexed W-types supports the theory of IIT signatures, and as a consequence all IITs. The difficulty in this construction is that the theory of IIT signatures is itself a QIIT, it is both inductive-inductive and has equality constructors. However, it can be seen as the well-typed syntax of a small type theory without any computation rules. Hence we can represent the syntax of normal forms without quotienting. We construct this well-typed normal syntax using preterms and typing relations from indexed W-types. Finally, we prove the elimination principle in the style of the initiality proof of Streicher.

Streicher [30] constructs the syntactic model of type theory using well-typed preterms and then shows initiality of this model by (1) defining a partial map to any other model by induction on preterms and (2) showing that whenever this partial function receives a well-typed preterm on its input it actually gives an output. Instead of defining a partial function, we define the graph of the same function as a relation and then show that it is functional as a second step. This can be seen as an indexed variant of the construction using partial functions.

Just as [21], we only consider finitary IITs, that is, constructors can only have a finite number of recursive arguments. An example constructor for `Con-Ty` which is not allowed is the following:

$$\Pi_{\infty} : (\Gamma : \text{Con}) \rightarrow (\mathbb{N} \rightarrow \text{Ty } \Gamma) \rightarrow \text{Ty } \Gamma$$

Structure of paper and list of contributions

We describe related work in Section 1.1, and explain our notation and Agda formalisation in Section 1.2. Then the following three sections describe our three contributions:

- Section 2. We define what it means for a model of extensional type theory (ETT, Definition 1) to support all inductive-inductive types (IITs): Definition 12. The novel contribution here is a (predicative) Church encoding of signatures following [8].
- Section 3. In Theorem 23, we show that if a model of ETT supports the theory of IIT signatures (Definition 15), then it supports IITs. This is an adaptation of a proof in [21].
- Section 4. Our main contribution is showing that if a model of ETT supports indexed W-types, then it supports the theory of IIT signatures (Theorem 57), and hence, all IITs (Corollary 58).

We list further work in Section 5.

The contents of this paper were presented at the TYPES 2019 conference in Oslo [22].

1.1 Related Work

The current work builds heavily on the work of Kaposi et al. [21] on finitary quotient inductive-inductive types (QIITs); we reuse both QIIT syntax and semantics by restricting to IITs, and we reuse the term model construction of QIITs as well. We also make use of the extension to infinitary QIITs [24] to derive the specification of the elimination principle for the theory of IIT signatures.

IITs (although not by this name) were first used to describe the well-typed syntax of type theory [15, 13]. Agda supported these general inductive definitions even before they were named IITs and given semantics by Nordvall Forsberg and Setzer [27]. Nordvall Forsberg's thesis [26] contains a specification similar in style to Dybjer and Setzer's codes for inductive-recursive types [17]. He also develops a categorical semantics based on dialgebras and provides a reduction of IITs to indexed inductive types, however only constructs the simple elimination principle as opposed to the general one. Altenkirch et al. [2] define signatures for QIITs (thus IITs as well) and their categorical semantics, however without proving existence of initial algebras. Their notion of signature, like Nordvall Forsberg's, involves more encoding overhead than ours.

Cartmell [12] introduced generalised algebraic theories using a type-theoretic syntax. Removing equations from his signatures and only considering finite signatures, we obtain finitary IIT signatures similar to ours. He does not consider constructing initial algebras using simpler classes of inductive types.

Hugunin [19] constructs several IITs in cubical Agda from inductive types. In this setting, the lack of UIP makes constructions significantly more involved, and essentially involves coinductive-coinductive well-formedness predicates defined as homotopy limits. Hugunin does not consider a generic syntax of IITs and only works on specific examples (although the examples vary greatly). He also only constructs simple elimination principles.

Streicher [30] presents an interpretation of the well-formed presyntax of a type theory into a categorical model, which is an important ingredient in constructing an initial model, although he does not present details on the construction of the term model or its initiality proof. Our initiality proof can be seen as an indexed variant of his construction (see Subsection 4.2 for a comparison).

Voevodsky was interested in constructing initial models of type theories from presyntaxes. Inspired by this, Brunerie et al. [10] formalised Streicher's proof in Agda for a type theory with Π , Σ , \mathbb{N} , identity types and an infinite hierarchy of universes. They used UIP, function extensionality and quotient types in the formalisation. In this paper we construct a type theory without computation rules, hence we avoid using quotients.

Intrinsic (well-typed) syntaxes for type theories were constructed using IITs [13], inductive-recursive types [15, 6] and QIITs [4]. In this paper we avoid using such general classes of inductive types as our goal is to reduce IITs to indexed inductive types.

Reducing general classes of inductive types to simpler classes has a long tradition in type theory. Indexed W-types were reduced to W-types [3] (using the essentially Streicher's idea of preterms and a typing predicate), small inductive-recursive types to indexed W-types [25], mutual inductive types to indexed W-types [23], W-types to natural numbers and quotients [1]. (Q)IITs can be reduced to quotient inductive types using the reduction of generalised algebraic theories to essentially algebraic theories [12]. Using the same reduction as mutual inductive types to indexed inductive types, (Q)IITs with more than two sorts can be reduced to (Q)IITs with only two sorts [20].

Awodey, Frey and Speight [8] construct inductive types using a restricted Church encoding in a type theory with an impredicative universe. We use the predicative version of their encoding to define IIT signatures.

Our reduction of IITs to indexed inductive types goes through two steps: first we construct a concrete QIIT using inductive types, then we construct all IITs from this particular QIIT. A more direct approach is proposed by [5]: here the initial algebra would be constructed directly for any IIT signature without going through an intermediate step.

1.2 Notation and Formalisation

► **Definition 1** (Model of extensional type theory (ETT)). *By a model of ETT we mean a category with families (CwF) [16, 18] with a countable predicative hierarchy of universes closed under the following type formers: Π , Σ , \top and an identity type with uniqueness of identity proofs and equality reflection.*

We will use Agda-like type theoretic syntax to work in the internal language of models of ETT:

- Universes are written Set_i . We usually omit level indices in this paper.
- Π types are notated as $(x : A) \rightarrow B$, or as $A \rightarrow B$ when non-dependent. We sometimes omit function arguments, by implicitly generalising over variables.
- Σ -types, notated either as $(x : A) \times B$, or as $\sum_x B$ when we want to leave the type of the first projection implicit. Projections are either named or given by proj_1 and proj_2 . We use $A \times B$ for non-dependent pairs.
- The unit type \top has the constructor tt which is definitionally equal to all elements of \top .
- The equality (identity) type is written $t = u$, it has a constructor $\text{refl} : t = t$, and equality reflection, hence we use the same $=$ sign for definitional equality. We occasionally indicate by $e_1, \dots, e_n \# t$ that t is well-typed thanks to the equalities e_1, \dots, e_n . To construct proofs, sometimes we write equational reasoning, e.g. $fa \stackrel{e}{=} fb$ where $e : a = b$. We also have uniqueness of identity proofs (UIP), expressing $(e : t = t) \rightarrow e = \text{refl}$. Note that function extensionality, expressing $((x : A) \rightarrow f x = g x) \rightarrow f = g$ is derivable.

The contents of Section 4 were formalised in Agda, the formalisation is available at <https://github.com/ambfont/UniversalIII>. Agda’s pattern matching mechanism implies uniqueness of identity proofs, we assumed function extensionality as an axiom and used rewrite rules [14] to obtain limited equality reflection.

2 A Definition of Inductive-Inductive Types

In this section we specify what it means that a model of ETT supports IITs. We first define the notion of IIT signature. Signatures for algebraic theories are usually given by inductive definitions. On the one hand, we take this even further: our notion of signature is given by a small type theory tailor-made to describe signatures, which we call the *theory of IIT signatures*. On the other hand we would like to avoid using a complicated inductive definition (a type theory is a quotient inductive-inductive type [4]) to describe a simpler class of inductive types. Hence we use a Church encoding [8] of the theory of IIT signatures, thereby avoiding the need for pre-existing inductive definitions. Another feature of our signatures is that they can include types from the model of ETT (such as \mathbb{N} in the `isEven–isOdd`). This is why signatures are specified internally to the particular model of ETT.²

We define the theory of IIT signatures by saying what its algebras (models) are. We call the *theory of IIT signatures algebras* simply *signature algebras*. The theory of signatures is a small type theory consisting of a (1) a substitution calculus (category with families, CwF [16]) equipped with (2) a universe, (3) a function space where the domain is in the universe and (4) another function space with external domain. We explain the usage of these type formers through examples after the definition.

² There is another method inspired by Capriotti [11] which allows stating what it means that any CwF \mathcal{C} (not necessarily a model of ETT) supports IITs with definitional computation rules. In this method, signatures are described in the internal language of $\hat{\mathcal{C}}$, the presheaf model over \mathcal{C} . We do not use this approach because it is more technical, and it would not strengthen our main result Corollary 58 as the proof of Theorem 57 needs \mathcal{C} to be a model of ETT.

► **Definition 2** (Signature algebra, SignAlg). *In a model of ETT, a signature algebra is an iterated Σ type consisting of the following four (families of) sets, 17 operations and 18 equalities.*

(1) *Substitution calculus*

$$\begin{aligned}
\text{Con} & : \text{Set} \\
\text{Ty} & : \text{Con} \rightarrow \text{Set} \\
\text{Sub} & : \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \\
\text{Tm} & : (\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Set} \\
\text{id} & : \text{Sub } \Gamma \Gamma \\
-\circ- & : \text{Sub } \Theta \Delta \rightarrow \text{Sub } \Gamma \Theta \rightarrow \text{Sub } \Gamma \Delta \\
\text{ass} & : (\sigma \circ \delta) \circ \nu = \sigma \circ (\delta \circ \nu) \\
\text{idl} & : \text{id} \circ \sigma = \sigma \\
\text{idr} & : \sigma \circ \text{id} = \sigma \\
-[-] & : \text{Ty } \Delta \rightarrow \text{Sub } \Gamma \Delta \rightarrow \text{Ty } \Gamma \\
-[-] & : \text{Tm } \Delta A \rightarrow (\sigma : \text{Sub } \Gamma \Delta) \rightarrow \text{Tm } \Gamma (A[\sigma]) \\
[\text{id}] & : A[\text{id}] = A \\
[\circ] & : A[\sigma \circ \delta] = A[\sigma][\delta] \\
[\text{id}] & : t[\text{id}] = t \\
[\circ] & : t[\sigma \circ \delta] = t[\sigma][\delta] \\
\cdot & : \text{Con} \\
\epsilon & : \text{Sub } \Gamma \cdot \\
\cdot\eta & : (\sigma : \text{Sub } \Gamma \cdot) \rightarrow \sigma = \epsilon \\
-\triangleright- & : (\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Con} \\
-, - & : (\sigma : \text{Sub } \Gamma \Delta) \rightarrow \text{Tm } \Gamma (A[\sigma]) \rightarrow \text{Sub } \Gamma (\Delta \triangleright A) \\
\pi_1 & : \text{Sub } \Gamma (\Delta \triangleright A) \rightarrow \text{Sub } \Gamma \Delta \\
\pi_2 & : (\sigma : \text{Sub } \Gamma (\Delta \triangleright A)) \rightarrow \text{Tm } \Gamma (A[\pi_1 \sigma]) \\
\pi_1 \beta & : \pi_1(\sigma, t) = \sigma \\
\pi_2 \beta & : \pi_2(\sigma, t) = t \\
\pi \eta & : (\pi_1 \sigma, \pi_2 \sigma) = \sigma \\
, \circ & : (\sigma, t) \circ \delta = (\sigma \circ \delta, t[\delta])
\end{aligned}$$

(2) *Universe*

$$\begin{aligned}
\text{U} & : \text{Ty } \Gamma \\
\text{El} & : \text{Tm } \Gamma \text{U} \rightarrow \text{Ty } \Gamma \\
\text{U}[] & : \text{U}[\sigma] = \text{U} \\
\text{El}[] & : (\text{El } a)[\sigma] = \text{El}(a[\sigma])
\end{aligned}$$

(3) *Inductive parameters*

$$\begin{aligned}
\Pi & : (a : \text{Tm } \Gamma \text{U}) \rightarrow \text{Ty } (\Gamma \triangleright \text{El } a) \rightarrow \text{Ty } \Gamma \\
-\@- & : \text{Tm } \Gamma (\Pi a B) \rightarrow (u : \text{Tm } \Gamma (\text{El } a)) \rightarrow \text{Tm } \Gamma (\text{El}(B[\text{id}, u])) \\
\Pi[] & : (\Pi a B)[\sigma] = \Pi(a[\sigma])(B[\sigma \circ \text{p}, \text{q}]) \\
\@[] & : (t \@ \alpha)[\sigma] = (t[\sigma]) \@ (\alpha[\sigma])
\end{aligned}$$

6:8 For Finitary Induction-Induction, Induction Is Enough

(4) *External parameters*

$$\begin{aligned} \hat{\Pi} & : (T : \text{Set}) \rightarrow (T \rightarrow \text{Ty } \Gamma) \rightarrow \text{Ty } \Gamma \\ -\hat{\alpha} - & : \text{Tm } \Gamma (\hat{\Pi} T B) \rightarrow (\alpha : T) \rightarrow \text{Tm } \Gamma (B \alpha) \\ \hat{\Pi}[] & : (\hat{\Pi} T B)[\sigma] = \hat{\Pi} T (\lambda \alpha. (B \alpha)[\sigma]) \\ \hat{\alpha}[] & : (t \hat{\alpha} \alpha)[\sigma] = (t[\sigma]) \hat{\alpha} \alpha \end{aligned}$$

Given an $M : \text{SignAlg}$, we denote its components by Con^M , Ty^M , Sub^M , Tm^M , id^M , and so on. We omit the indices if there is only one signature algebra in scope (e.g. in Definition 3 and Example 4).

► **Definition 3** (Abbreviations). For a signature algebra, we use $\text{wk} : \text{Sub}(\Gamma \triangleright A) \Gamma$ to mean $\pi_1 \text{id}$. We recover de Bruijn indices by setting $0 := \pi_2 \text{id}$ and $1 + n := n[\text{wk}]$. $\Pi a (B[\text{wk}])$ is abbreviated by $a \Rightarrow B$, $\hat{\Pi} T (\lambda _ . B)$ by $T \Rightarrow B$.

► **Example 4** (Example contexts in a signature algebra). Given a signature algebra, we can define a context which specifies natural numbers. For readability, an informal version of the same context is displayed on the right using variable names.

$$\cdot \triangleright \text{U} \triangleright z : \text{El } 0 \triangleright s : 1 \Rightarrow \text{El } 1 \qquad \cdot \triangleright N : \text{U} \triangleright z : \text{El } N \triangleright s : N \Rightarrow \text{El } N$$

We start with the empty context \cdot , then we declare a sort U , then we declare an operator producing an element of the sort denoted by $\text{El } 0$ where 0 is the de Bruijn index referring to the sort. Finally, we declare an operator which takes as input an element of the sort (now it became de Bruijn index 1) and produces an element of the same sort. Note the asymmetry of the function type \Rightarrow : the domain needs to be an element of U , while the codomain can be any type (including another function type). This ensures strict positivity of the operators.

Lists with elements of a given $T : \text{Set}$ type are given by the following context. Here we use the function space with external domain \Rightarrow to include a T in the signature. For readability, we omit the λ and the superscripts and we do not write the compatibility condition. On the right we list the same signature with variable names.

$$\cdot \triangleright \text{U} \triangleright \text{El } 0 \triangleright T \Rightarrow 1 \Rightarrow \text{El } 1 \qquad \cdot \triangleright L : \text{U} \triangleright \text{nil} : \text{El } L \triangleright \text{cons} : T \Rightarrow L \Rightarrow \text{El } L$$

The Con – Ty example from Section 1 is given by the following context.

$$\begin{array}{ll} \cdot \triangleright & \cdot \triangleright \\ \text{U} \triangleright & \text{Con} : \text{U} \triangleright \\ 0 \Rightarrow \text{U} \triangleright & \text{Ty} : \text{Con} \Rightarrow \text{U} \triangleright \\ \text{El } 1 \triangleright & \text{empty} : \text{El } \text{Con} \triangleright \\ \Pi 2 (2 @ 0 \Rightarrow \text{El } 3) \triangleright & \text{ext} : \Pi (\Gamma : \text{Con}) (\text{Ty} @ \Gamma \Rightarrow \text{El } \text{Con}) \triangleright \\ \Pi 3 (\text{El } (3 @ 0)) \triangleright & \text{U} : \Pi (\Gamma : \text{Con}) (\text{El } (\text{Ty} @ \Gamma)) \triangleright \\ \Pi 4 (\text{El } (4 @ (2 @ 0 @ (1 @ 0)))) & \text{El} : \Pi (\Gamma : \text{Con}) (\text{El } (\text{Ty} @ (\text{ext} @ \Gamma @ (\text{U} @ \Gamma)))) \end{array}$$

The above examples are contexts in any signature algebra, and we could take this as a definition of signature: $(M : \text{SignAlg}) \rightarrow \text{Con}^M$ is the usual Church-encoding of contexts. However (as we will see in Remark 24) the notion of constructor for such signatures would be too strong. Another approach would be to assume that there is a syntax for signature

algebras (an initial signature algebra), and then a signature would be a context in this signature algebra. We will define syntactic signatures using this approach in the next section (Definition 16), but for now we do not want to assume the existence of any inductive type. Instead, we will use a restricted Church encoding. This requires the notion of morphism of signatures.

The notion of morphism is determined by the notion of algebra [24], but we include it here for completeness.

► **Definition 5** (Signature morphism, SignMor). *A morphism from signature algebras M to N denoted $\text{SignMor } M \ N$ consists of four functions and 17 equalities expressing that the functions preserve the operations of the two algebras. We use the same naming as in Definition 2 and use superscripts to denote which algebra is meant.*

(1) *Substitution calculus*

$$\begin{aligned}
\text{Con} &: \text{Con}^M && \rightarrow \text{Con}^N \\
\text{Ty} &: \text{Ty}^M \Gamma && \rightarrow \text{Ty}^N (\text{Con } \Gamma) \\
\text{Sub} &: \text{Sub}^M \Gamma \Delta && \rightarrow \text{Sub}^N (\text{Con } \Gamma) (\text{Con } \Delta) \\
\text{Tm} &: \text{Tm}^M \Gamma A && \rightarrow \text{Tm}^N (\text{Con } \Gamma) (\text{Ty } A) \\
\text{id} &: \text{Sub id}^M && = \text{id}^N \\
\circ &: \sigma \circ^M \delta && = \text{Sub } \sigma \circ^N \text{Sub } \delta \\
[] &: A[\sigma]^M && = \text{Ty } A[\text{Sub } \sigma]^N \\
[] &: t[\sigma]^M && = \text{Tm } t[\text{Sub } \sigma]^N \\
\cdot &: \text{Con } \cdot^M && = \cdot^N \\
\epsilon &: \text{Sub } \epsilon^M && = \epsilon^N \\
\triangleright &: \text{Con } (\Gamma \triangleright^M A) && = \text{Con } \Gamma \triangleright^N \text{Ty } A \\
, &: \text{Sub } (\sigma,^M t) && = \text{Sub } \sigma,^N \text{Tm } t \\
\pi_1 &: \text{Sub } (\pi_1^M \sigma) && = \pi_1^N (\text{Sub } \sigma) \\
\pi_2 &: \text{Tm } (\pi_2^M \sigma) && = \pi_2^N (\text{Sub } \sigma)
\end{aligned}$$

(2) *Universe*

$$\begin{aligned}
\text{U} &: \text{Ty } \text{U}^M && = \text{U}^N \\
\text{El} &: \text{Ty } (\text{El}^M a) && = \text{El}^N (\text{Tm } a)
\end{aligned}$$

(3) *Inductive parameters*

$$\begin{aligned}
\Pi &: \text{Ty } (\Pi^M a B) && = \Pi^N (\text{Tm } a) (\text{Ty } B) \\
\textcircled{\ast} &: \text{Tm } (t \textcircled{\ast}^M u) && = \text{Tm } t \textcircled{\ast}^N \text{Tm } u
\end{aligned}$$

(4) *External parameters*

$$\begin{aligned}
\hat{\Pi} &: \text{Ty } (\hat{\Pi}^M T B) && = \hat{\Pi}^N T (\lambda \alpha. \text{Ty } (B \alpha)) \\
\hat{\textcircled{\ast}} &: \text{Tm } (t \hat{\textcircled{\ast}}^M \alpha) && = \text{Tm } t \hat{\textcircled{\ast}}^N \alpha
\end{aligned}$$

Given an $f : \text{SignMor } M \ N$, we denote its first four components just by f_{Con} , f_{Ty} , f_{Sub} , f_{Tm} or just write f if it is clear which one is meant.

We define IIT signatures using the Church encoding introduced by Awodey, Frey and Speight [8]. A difference is that we avoid impredicativity. This restricts the possible eliminations on signatures: we can only eliminate into a universe which is smaller than the level of signatures. However, this still covers all eliminations in this paper, and it is also not an issue for us that signatures do not live in the smallest universe.

6:10 For Finitary Induction-Induction, Induction Is Enough

► **Definition 6** (IIT signature). *An IIT signature is a context in an arbitrary signature algebra, which is also compatible with morphisms:*

$$\begin{aligned} \text{Sign} := & (\text{sig} : (M : \text{SignAlg}) \rightarrow \text{Con}^M) \times \\ & ((M N : \text{SignAlg})(f : \text{SignMor } M N) \rightarrow f_{\text{Con}}(\text{sig } M) = \text{sig } N). \end{aligned}$$

The compatibility condition says that if we obtain an M -context using sig at signature algebra M and then we transport it to N using f , we get the same N -context as directly applying sig to N .

The lack of impredicativity implies that our notion of signatures do not form a signature algebra.

► **Lemma 7.** *There is no $M : \text{SignAlg}$, in which $\text{Con}^M = \text{Sign}$.*

Proof. If the Con component in SignAlg is Set_i , then SignAlg is in Set_{i+1} , but as Sign is defined as $(\text{SignAlg} \rightarrow \dots) \times \dots$, it is at least in Set_{i+1} , so we can't choose $\text{Con}^M : \text{Set}_i$ to be $\text{Sign} : \text{Set}_{i+1}$. ◀

Note that the notion of IIT signature is relative to a model of ETT: it is expressed as a term (of a function type) in the model. This is necessary because of the function space $\hat{\Pi}$, which has as domain an arbitrary type in the model. We make use of $\hat{\Pi}$ in signatures with external parameters, like the type of the elements in lists.

► **Example 8** (Example signature). Now we can formally describe the contexts given in Example 4 as signatures. For natural numbers, we have the following pair of functions. The second function returns an equality proof which we describe using equational reasoning.

$$\begin{aligned} (\text{nat}, \text{natc}) := & \\ & (\lambda M. (\cdot^M \triangleright^M \mathbf{U}^M \triangleright^M \text{El}^M 0^M \triangleright^M 1^M \Rightarrow^M \text{El}^M 1^M), \\ & \lambda M N f. f_{\text{Con}}(\cdot^M \triangleright^M \mathbf{U}^M \triangleright^M \text{El}^M 0^M \triangleright^M 1^M \Rightarrow^M \text{El}^M 1^M) = \\ & \quad f_{\text{Con}}(\cdot^M \triangleright^M \mathbf{U}^M \triangleright^M \text{El}^M 0^M) \triangleright^N f_{\text{Ty}}(1^N \Rightarrow^N \text{El}^N 1^N) = \\ & \quad f_{\text{Con}}(\cdot^M \triangleright^M \mathbf{U}^M) \triangleright^N f_{\text{Ty}}(\text{El}^M 0^M) \triangleright^N f_{\text{Tm}} 1^N \Rightarrow^M f_{\text{Ty}}(\text{El}^N 1^N) = \\ & \quad f_{\text{Con}} \cdot^M \triangleright^N f_{\text{Ty}} \mathbf{U}^M \triangleright^N \text{El}^N (f_{\text{Tm}} 0^M) \triangleright^N 1^M \Rightarrow^M \text{El}^M (f_{\text{Tm}} 1^N) = \\ & \quad \cdot^N \triangleright^N \mathbf{U}^N \triangleright^N \text{El}^N 0^N \triangleright^N 1^N \Rightarrow^N \text{El}^N 1^N) \end{aligned}$$

The first component builds the context describing natural numbers in M , the second one uses the fact that f is a morphism, that is, it preserves all operations.

The signatures for lists and Con-Ty can be given analogously.

Given a model of ETT and an IIT signature in it, we would like to say what it means that the model supports the given IIT. For this we define the signature algebra ADS which will provide notions of algebras, displayed algebras and sections for each signature. This is the same as the $-^A$, $-^D$ and $-^S$ operations in [21]. Before defining ADS , we illustrate its usage by an example.

► **Example 9** (Algebras, displayed algebras and sections for natural numbers). For the signature of natural numbers as given in Example 8, algebras are given by the Σ -type $(N : \text{Set}) \times N \times (N \rightarrow N)$. A displayed algebra over (N, z, s) is given by the Σ -type

$$(N^D : N \rightarrow \text{Set}) \times N^D z \times ((n : N) \rightarrow N^D n \rightarrow N^D (s n)),$$

and a section of a displayed algebra (N^D, z^D, s^D) over (N, z, s) is given by the Σ -type

$$(N^S : (n : N) \rightarrow N^D n) \times (N^S z = z^D) \times ((n : N) \rightarrow N^S (s n) = s^D n (N^S n)).$$

Displayed algebras over the initial algebra are called motives and methods of the eliminator, while a section of a displayed algebra over the initial algebra is the eliminator together with its computation rules.

► **Definition 10** (The signature algebra ADS). *We define an element of SignAlg by listing all its components Con, Ty, Sub, and so on, one per row. Each such component has three parts denoted by A , D and S , respectively. The equality components of SignAlg are omitted as they are all reflexivity.*

$$\begin{array}{lll}
(\Gamma^A : \mathbf{Set}) & \times (\Gamma^D : \Gamma^A \rightarrow \mathbf{Set}) & \times (\Gamma^S : (\gamma : \Gamma^A) \rightarrow \Gamma^D \gamma \rightarrow \mathbf{Set}) \\
(A^A : \Gamma^A \rightarrow \mathbf{Set}) & \times (A^D : \Gamma^D \gamma \rightarrow A^A \gamma \rightarrow \mathbf{Set}) & \times (A^S : \Gamma^S \gamma \gamma^D \rightarrow (\alpha : A^A \gamma) \rightarrow \\
& & A^D \gamma^D \alpha \rightarrow \mathbf{Set}) \\
(\sigma^A : \Gamma^A \rightarrow \Delta^A) & \times (\sigma^D : \Gamma^D \gamma \rightarrow \Delta^D (\sigma^A \gamma)) & \times (\sigma^S : \Gamma^S \gamma \gamma^D \rightarrow \\
& & \Delta^S (\sigma^A \gamma) (\sigma^D \gamma^D)) \\
(t^A : (\gamma : \Gamma^A) \rightarrow A^A \gamma) & \times (t^D : (\gamma^D : \Gamma^D \gamma) \rightarrow \\
& A^D \gamma^D (t^A \gamma)) & \times (t^S : (\gamma^S : \Gamma^S \gamma \gamma^D) \rightarrow \\
& & A^S (t^A \gamma) (t^D \gamma^D)) \\
\text{id}^A \gamma := \gamma & \text{id}^D \gamma^D := \gamma^D & \text{id}^S \gamma^S := \gamma^S \\
(\sigma \circ \delta)^A \gamma := \sigma^A (\delta^A \gamma) & (\sigma \circ \delta)^D \gamma^D := \sigma^D (\delta^D \gamma^D) & (\sigma \circ \delta)^S \gamma^S := \sigma^S (\delta^S \gamma^S) \\
(A[\sigma])^A \gamma := A^A (\sigma^A \gamma) & (A[\sigma])^D \gamma^D := A^D (\sigma^D \gamma^D) & (A[\sigma])^S \gamma^S := A^S (\sigma^S \gamma^S) \\
(t[\sigma])^A \gamma := t^A (\sigma^A \gamma) & (t[\sigma])^D \gamma^D := t^D (\sigma^D \gamma^D) & (t[\sigma])^S \gamma^S := t^S (\sigma^S \gamma^S) \\
\cdot^A := \top & \cdot^D _ := \top & \cdot^S _ := \top \\
\epsilon^A _ := \text{tt} & \epsilon^D _ := \text{tt} & \epsilon^S _ := \text{tt} \\
(\Gamma \triangleright A)^A := & (\Gamma \triangleright A)^D (\gamma, \alpha) := & (\Gamma \triangleright A)^S (\gamma, \alpha) (\gamma^D, \alpha^D) := \\
(\gamma : \Gamma^A) \times A^A \gamma & (\gamma^D : \Gamma^D \gamma) \times A^D \gamma^D \alpha & (\gamma^S : \Gamma^S \gamma \gamma^D) \times A^S \gamma^S \alpha \alpha^D \\
(\sigma, t)^A \gamma := (\sigma^A \gamma, t^A \gamma) & (\sigma, t)^D \gamma^D := (\sigma^D \gamma^D, t^D \gamma^D) & (\sigma, t)^S \gamma^S := (\sigma^S \gamma^S, t^S \gamma^S) \\
(\pi_1 \sigma)^A \gamma := \text{proj}_1 (\sigma^A \gamma) & (\pi_1 \sigma)^D \gamma^D := \text{proj}_1 (\sigma^D \gamma^D) & (\pi_1 \sigma)^S \gamma^S := \text{proj}_1 (\sigma^S \gamma^S) \\
(\pi_2 \sigma)^A \gamma := \text{proj}_2 (\sigma^A \gamma) & (\pi_2 \sigma)^D \gamma^D := \text{proj}_2 (\sigma^D \gamma^D) & (\pi_2 \sigma)^S \gamma^S := \text{proj}_2 (\sigma^S \gamma^S) \\
\mathbf{U}^A \gamma := \mathbf{Set} & \mathbf{U}^D \gamma^D T := T \rightarrow \mathbf{Set} & \mathbf{U}^S \gamma^S T T^D := (\alpha : T) \rightarrow T^D \alpha \\
(\text{El } a)^A \gamma := a^A \gamma & (\text{El } a)^D \gamma^D \alpha := a^D \gamma^D \alpha & (\text{El } a)^S \gamma^S \alpha \alpha^D := (a^S \gamma^S \alpha = \alpha^D) \\
(\Pi a B)^A \gamma := & (\Pi a B)^D \gamma^D f := & (\Pi a B)^S \gamma^S f f^D := (\alpha : a^A \gamma) \rightarrow \\
(\alpha : a^A \gamma) \rightarrow B^A (\gamma, \alpha) & (\alpha^D : a^D \gamma^D \alpha) \rightarrow & B^S (\gamma^S, \text{refl}_{a^S \gamma^S \alpha}) (f \alpha) \\
& B^D (\gamma^D, \alpha^D) (f \alpha) & (f^D (a^S \gamma^S \alpha)) \\
(t \otimes u)^A \gamma := t^A \gamma (u^A \gamma) & (t \otimes u)^D \gamma^D := t^D \gamma^D (u^D \gamma^D) & (t \otimes u)^S \gamma^S :=_{u^S \gamma^S \#} t^S \gamma^S (u^A \gamma) \\
(\hat{\Pi} T B)^A \gamma := & (\hat{\Pi} T B)^D \gamma^D f := & (\hat{\Pi} T B)^S \gamma^S f f^D := (\alpha : T) \rightarrow \\
(\alpha : T) \rightarrow (B \alpha)^A \gamma & (\alpha : T) \rightarrow (B \alpha)^D \gamma^D (f \alpha) & (B \alpha)^S \gamma^S (f \alpha) (f^D \alpha) \\
(t \hat{\otimes} \alpha)^A \gamma := t^A \gamma \alpha & (t \hat{\otimes} \alpha)^D \gamma^D := t^D \gamma^D \alpha & (t \hat{\otimes} \alpha)^S \gamma^S := t^S \gamma^S \alpha
\end{array}$$

Definition 10 can be explained by columns (see [21, Sections 4 and 6] for more details) or by rows (see [21, Section 7.4]).

We first explain it by columns: the first column (A components) corresponds to the standard model (set model, metacircular interpretation [4]): contexts are sets, types are families, terms are functions, the universe \mathbf{U} is given by \mathbf{Set} , function spaces are given by the external function space. The D column is a logical predicate interpretation, A and D together are a unary version of the parametric model for dependent types [7]. Contexts are predicates,

6:12 For Finitary Induction-Induction, Induction Is Enough

types are families of predicates, terms say that the \hat{A} interpretation respects the predicates (this is usually called fundamental lemma of the logical predicate). \mathbf{U} is given by predicate space, the predicate at a Π type holds for a function if it respects the predicates. For $\hat{\Pi}$, the predicate is defined pointwise. The last column \mathbf{S} is a modified dependent logical relation which refers to both \hat{A} and \hat{D} . Contexts are binary relations where the second parameter depends on the first one, types are dependent variants of this, terms say that the relation is respected by \hat{A} and \hat{D} , respectively. \mathbf{U} is however not relation space, but a function and $(\text{El } a)^{\mathbf{S}}$ is the graph of the function $a^{\mathbf{S}}$. $\Pi^{\mathbf{S}}$ for a function again says that the function respects the relation, however we do not simply say

$$(\Pi a B)^{\mathbf{S}} \gamma^{\mathbf{S}} f f^D := (\alpha : a^{\hat{A}} \gamma)(\alpha^D : a^{\hat{D}} \gamma^D \alpha)(\alpha^{\mathbf{S}} : (\text{El } a)^{\mathbf{S}} \gamma^{\mathbf{S}} \alpha \alpha^D) \rightarrow B^{\mathbf{S}} \dots,$$

as $(\text{El } a)^{\mathbf{S}} \gamma^{\mathbf{S}} \alpha \alpha^D$ is just an equality $a^{\mathbf{S}} \gamma^{\mathbf{S}} \alpha = \alpha^D$ which we can singleton contract. So we omit α^D and this equality as an input and replace α^D by $a^{\mathbf{S}} \gamma^{\mathbf{S}} \alpha$ in the definition.

When viewing ADS by rows, we can see that it is a part of the CwF model of type theory [21, Section 7.4]. In the CwF model, a context is given by a CwF. Now, from the category part of the CwF, we only have objects $(\Gamma^{\hat{A}})$, and from the families, we have the families for types $\Gamma^{\hat{D}}$ and terms $\Gamma^{\mathbf{S}}$. Types are the corresponding parts of displayed CwFs, substitutions are parts of CwF morphisms, terms are parts of CwF sections. \mathbf{U} is part of the CwF of sets, $\text{El } a$ is the part of the discrete displayed CwF coming from a (which is a CwF-morphism from Γ to the CwF of sets). Π is given by a dependent product of displayed CwFs where it is essential that the domain is discrete, $\hat{\Pi}$ is the pointwise direct product.

► **Definition 11** (The set signature algebra \mathbf{A}). $\mathbf{A} : \text{SignAlg}$ is given by the first \hat{A} components of ADS (Definition 10), that is, $\text{Con}^{\hat{A}} := \text{Set}$, $\text{Ty}^{\hat{A}} \Gamma := \Gamma \rightarrow \text{Set}$, $\text{Sub}^{\hat{A}} \Gamma \Delta := \Gamma \rightarrow \Delta$, and so on. There is a morphism from ADS to \mathbf{A} defined by $-^{\hat{A}}$ at each component, which we also denote by $-^{\hat{A}} : \text{SignMor ADS } \mathbf{A}$.

► **Definition 12** (A model of ETT supports IITs). A model of ETT supports IITs if for any signature $(\text{sig}, \text{sigc}) : \text{Sign}$ there is a

$$\text{con}_{\text{sig}} : (\text{sig ADS})^{\hat{A}}$$

and an

$$\text{elim}_{\text{sig}} : (\gamma^D : (\text{sig ADS})^{\hat{D}} \text{con}_{\text{sig}}) \rightarrow (\text{sig ADS})^{\mathbf{S}} \text{con}_{\text{sig}} \gamma^D.$$

In other words, for any signature, we have an algebra called con (constructors) and for any displayed algebra over the constructors, we have a section (called the eliminator).

One can check that Definition 12 gives the right notion of constructors and elimination principle for the signatures in Example 8.

► **Example 13** (A model of ETT supports natural numbers). For the signature $(\text{nat}, \text{natc})$ of natural numbers in Example 8, the type of con_{nat} is

$$\begin{aligned} (\text{nat ADS})^{\hat{A}} &= \\ (\cdot^{\text{ADS}} \triangleright^{\text{ADS}} \mathbf{U}^{\text{ADS}} \triangleright^{\text{ADS}} \text{El}^{\text{ADS}} 0^{\text{ADS}} \triangleright^{\text{ADS}} 1^{\text{ADS}} \Rightarrow^{\text{ADS}} \text{El}^{\text{ADS}} 1^{\text{ADS}})^{\hat{A}} &= \\ \left(((\cdot \triangleright \mathbf{U}) \triangleright \text{El}(\pi_2 \text{id})) \triangleright (\pi_2(\pi_1 \text{id})) \Rightarrow \text{El}(\pi_2(\pi_1 \text{id})) \right)^{\hat{A}} &= \\ \left(\gamma'' : (\gamma' : ((\gamma : \hat{A}) \times \mathbf{U}^{\hat{A}} \gamma)) \times (\text{El}(\pi_2 \text{id}))^{\hat{A}} \gamma' \right) \times \left(\Pi(\pi_2(\pi_1 \text{id})) (\pi_2(\pi_1(\pi_1 \text{id}))) \right)^{\hat{A}} \gamma'' &= \\ \left(\gamma'' : (\gamma' : ((\gamma : \top) \times \text{Set})) \times (\text{proj}_2 \gamma') \right) \times (\text{proj}_2(\text{proj}_1 \gamma'') \rightarrow \text{proj}_2(\text{proj}_1 \gamma'')), & \end{aligned}$$

which is a left-nested Σ type isomorphic to its right-nested counterpart

$$(N : \text{Set}) \times (N \times (N \rightarrow N)).$$

Writing $((\text{tt}, \text{Nat}), \text{zero}), \text{suc}$ for con_{nat} , the type of elim_{nat} computes as follows.

$$\begin{aligned} & (\gamma^D : (\text{nat ADS})^D \text{con}_{\text{nat}}) \rightarrow (\text{nat ADS})^S \text{con}_{\text{nat}} \gamma^D = \\ & \left(\gamma^D : \left(((\cdot \triangleright \text{U}) \triangleright \text{El}(\pi_2 \text{id})) \triangleright (\pi_2(\pi_1 \text{id})) \Rightarrow \text{El}(\pi_2(\pi_1 \text{id}))) \right)^D \text{con}_{\text{nat}} \right) \rightarrow \\ & \left(((\cdot \triangleright \text{U}) \triangleright \text{El}(\pi_2 \text{id})) \triangleright (\pi_2(\pi_1 \text{id})) \Rightarrow \text{El}(\pi_2(\pi_1 \text{id})) \right)^S \text{con}_{\text{nat}} \gamma^D = \\ & \left(\gamma^D : \left(((\cdot \triangleright \text{U}) \triangleright \text{El}(\pi_2 \text{id})) \triangleright (\pi_2(\pi_1 \text{id})) \Rightarrow \text{El}(\pi_2(\pi_1 \text{id}))) \right)^D \right. \\ & \quad \left. ((\text{tt}, \text{Nat}), \text{zero}), \text{suc} \right) \rightarrow \\ & \left(((\cdot \triangleright \text{U}) \triangleright \text{El}(\pi_2 \text{id})) \triangleright (\pi_2(\pi_1 \text{id})) \Rightarrow \text{El}(\pi_2(\pi_1 \text{id})) \right)^S ((\text{tt}, \text{Nat}), \text{zero}), \text{suc} \gamma^D = \\ & \left((((\text{tt}, N^D), z^D), s^D) : \left(\gamma^{D''} : (\gamma^{D'} : ((\gamma^D : \cdot^D \text{tt}) \times \text{U}^D \gamma^D \text{Nat})) \times (\text{El}(\pi_2 \text{id}))^D \gamma^{D'} \text{zero} \right) \times \right. \\ & \quad \left. \left(\Pi(\pi_2(\pi_1 \text{id}))(\pi_2(\pi_1(\pi_1 \text{id}))) \right)^D \gamma^{D''} \text{suc} \right) \rightarrow \\ & \left(((\cdot \triangleright \text{U}) \triangleright \text{El}(\pi_2 \text{id})) \triangleright (\pi_2(\pi_1 \text{id})) \Rightarrow \text{El}(\pi_2(\pi_1 \text{id})) \right)^S ((\text{tt}, \text{Nat}), \text{zero}), \text{suc} \\ & \quad ((\text{tt}, N^D), z^D), s^D) = \\ & \left((((\text{tt}, N^D), z^D), s^D) : \left(\gamma^{D''} : (\gamma^{D'} : ((\gamma^D : \cdot^D \text{tt}) \times \text{U}^D \gamma^D \text{Nat})) \times (\text{El}(\pi_2 \text{id}))^D \gamma^{D'} \text{zero} \right) \times \right. \\ & \quad \left. \left(\Pi(\pi_2(\pi_1 \text{id}))(\pi_2(\pi_1(\pi_1 \text{id}))) \right)^D \gamma^{D''} \text{suc} \right) \rightarrow \\ & \left(\gamma^{S''} : (\gamma^{S'} : ((\gamma^S : \cdot^S \text{tt tt}) \times \text{U}^S \gamma^S \text{Nat } N^D)) \times (\text{El}(\pi_2 \text{id}))^S \gamma^{S'} \text{zero } z^D \right) \times \\ & \left(\Pi(\pi_2(\pi_1 \text{id}))(\pi_2(\pi_1(\pi_1 \text{id}))) \right)^S \gamma^{S''} \text{suc } s^D = \\ & \left((((\text{tt}, N^D), z^D), s^D) : \left(\gamma^{D''} : (\gamma^{D'} : ((\gamma^D : \top) \times (\text{Nat} \rightarrow \text{Set}))) \times \text{proj}_2 \gamma^{D'} \text{zero} \right) \times \right. \\ & \quad \left. \left(\text{proj}_2(\text{proj}_1 \gamma^{D''}) n \rightarrow \text{proj}_2(\text{proj}_1 \gamma^{D''})(\text{suc } n) \right) \right) \rightarrow \\ & \left(\gamma^{S''} : (\gamma^{S'} : ((\gamma^S : \top) \times ((n : \text{Nat}) \rightarrow N^D n))) \times \text{proj}_2 \gamma^{S'} \text{zero} = z^D \right) \times \\ & \left((n : \text{Nat}) \rightarrow \text{proj}_2(\text{proj}_1(\text{proj}_1 \gamma^{S''}))(\text{suc } n) = s^D(\text{proj}_2(\text{proj}_1(\text{proj}_1 \gamma^{S''})) n) \right) \end{aligned}$$

This is again a left-nested version of the expected elimination principle

$$\begin{aligned} & (N^D : \text{Nat} \rightarrow \text{Set})(z^D : N^D \text{zero})(s^D : (n : \text{Nat}) \rightarrow N^D n \rightarrow N^D(\text{suc } n)) \rightarrow \\ & (N^S : (n : \text{Nat}) \rightarrow N^D n) \times (N^S \text{zero} = z^D) \times ((n : \text{Nat}) \rightarrow N^S(\text{suc } n) = s^D(N^S n)) \end{aligned}$$

► Remark 14. The computation rules of the elimination principle are only expected up to the internal equality type, but as we work with a model of ETT, we also get them as definitional equalities by equality reflection.

3 Constructing all IITs from the Theory of IIT Signatures

In the previous section, using the notions of signature algebras and signature morphisms, we defined IIT signatures and what it means for a model of ETT to support all IITs. In this section we show that if a model of ETT supports the theory of IIT signatures, then it supports all IITs. Using the Church encoding of Definition 6, every model of ETT can describe IIT signatures. In contrast, in Definition 15, we will require existence of an initial signature algebra.

The contents of this section are an adjustment of [21, Sections 4 and 6] to our setting.

► **Definition 15.** *A model of ETT supports the theory of IIT signatures if there is a signature algebra $\mathsf{l} : \mathsf{SignAlg}$ equipped with a unique morphism $\llbracket - \rrbracket_M : \mathsf{SignMor} \mathsf{l} M$ into any algebra M . Sometimes we omit the subscript M . We call l the syntax or initial algebra, the morphism $\llbracket - \rrbracket$ is called recursor.*

► **Definition 16** (Syntactic signatures). *In a model of ETT supporting the theory of IIT signatures, we call elements of $\mathsf{Con}^{\mathsf{l}}$ syntactic signatures.*

One may wonder what is the relationship between the two notion of signatures.

► **Lemma 17.** *In a model of ETT supporting the theory of IIT signatures, signatures and syntactic signatures are isomorphic.*

Proof. We can turn a $(\mathit{sig}, \mathit{sigc}) : \mathsf{Sign}$ into $\mathsf{Con}^{\mathsf{l}}$ by $\mathit{sig} \mathsf{l}$ and an $\Omega : \mathsf{Con}^{\mathsf{l}}$ into a Sign by $(\lambda M. \llbracket \Omega \rrbracket_M, \lambda M N f. (f \llbracket \Omega \rrbracket_M = (f \circ \llbracket - \rrbracket_M) \Omega = \llbracket \Omega \rrbracket_N))$ where the equality proof in the second component comes from uniqueness of the recursor (we have to define composition of morphisms \circ for this). The compositions of these two maps are the identities: $(\mathit{sig}, \mathit{sigc})$ is mapped to $(\lambda M. \llbracket \mathit{sig} I \rrbracket_M, \dots) = (\lambda M. \llbracket - \rrbracket_M (\mathit{sig} I), \dots)$ which is equal to $(\lambda M. \mathit{sig} M, \dots)$ because of sigc ; Ω is mapped to $\llbracket \Omega \rrbracket_{\mathsf{l}} = \Omega$ by uniqueness of $\llbracket - \rrbracket$. ◀

We will define the term signature algebra by which we obtain the constructors con for any IIT signature. Then we will define another signature algebra which provides the eliminator. Before doing these, we illustrate the idea of both constructions on natural numbers.

► **Example 18.** For natural numbers, we will define the constructors con as the following natural number algebra $(\mathsf{Nat}, \mathit{zero}, \mathit{suc})$. We write variable names instead of de Bruijn indices for readability.

$$\mathsf{Nat} := \mathsf{Tm}^{\mathsf{l}} (\cdot \triangleright N : \mathsf{U} \triangleright z : \mathsf{El} N \triangleright s : N \Rightarrow \mathsf{El} N) (\mathsf{El} N)$$

$$\mathit{zero} := z$$

$$\mathit{suc} := \lambda t. (s @ t)$$

Natural numbers are simply l -terms of type $\mathsf{El} N$ in the context which is the syntactic signature for natural numbers. In this context, the only way to define a term of type $\mathsf{El} N$ is to use z and s , corresponding to the zero and suc constructors.

To define the action of the eliminator on a natural number $n : \mathsf{Nat}$, let's look at the type of the displayed algebra interpretation of the number:

$$\llbracket n \rrbracket_{\mathsf{ADS}}^{\mathsf{D}} : (\gamma^{\mathsf{D}} : \llbracket \cdot \triangleright N : \mathsf{U} \triangleright z : \mathsf{El} N \triangleright s : N \Rightarrow \mathsf{El} N \rrbracket^{\mathsf{D}} \mathsf{con}) \rightarrow \llbracket \mathsf{El} N \rrbracket^{\mathsf{D}} (\llbracket n \rrbracket^{\mathsf{A}} \mathsf{con})$$

This says that for a displayed algebra $\gamma^{\mathsf{D}} = (N^{\mathsf{D}}, z^{\mathsf{D}}, s^{\mathsf{D}})$ over con (i.e. the motives and methods of the eliminator), we get a witness of the predicate $\llbracket \mathsf{El} N \rrbracket^{\mathsf{D}} = N^{\mathsf{D}}$ at the algebra interpretation of n . This is not yet good, as we would like to get $N^{\mathsf{D}} n$ instead of $N^{\mathsf{D}} (\llbracket n \rrbracket^{\mathsf{A}} \mathsf{con})$ as a result. However, interpretation into the term signature algebra will imply that $n = \llbracket n \rrbracket^{\mathsf{A}} \mathsf{con}$.

► **Definition 19** (Term signature algebra IC_-). For an $\Omega : \text{Con}^I$, we define $\text{IC}_\Omega : \text{SignAlg}$ which we call the term signature algebra. It is equipped with a morphism $-^I : \text{SignMor}(\text{IC}_\Omega) \mathbb{I}$. We define IC_Ω by listing its components Con , Ty , Sub , and so on, one per row. Each component has two parts denoted by I and C . The I part just reuses the corresponding components from \mathbb{I} , and thus the morphism $-^I$ is defined as the obvious projection. We omit the equality components, as they come from UIP or are trivial. We also omit the components for terms and substitutions as their C parts consist of uninformative equational reasoning.

$$\begin{array}{ll}
\Gamma^I : \text{Con}^I & \Gamma^C : \text{Sub}^I \Omega \Gamma^I \rightarrow \llbracket \Gamma \rrbracket_A \\
A^I : \text{Ty}^I \Gamma^I & A^C : (\nu : \text{Sub}^I \Omega \Gamma^I) \rightarrow \text{Tm}^I \Omega (A^I[\nu]) \rightarrow \llbracket A \rrbracket_A (\Gamma^C \nu) \\
\sigma^I : \text{Sub}^I \Gamma^I \Delta^I & \sigma^C : \Delta^C (\sigma^I \circ \nu) = \llbracket \sigma \rrbracket_A (\Gamma^C \nu) \\
t^I : \text{Tm}^I \Gamma^I A^I & t^C : A^C \nu (t^I[\nu]) = \llbracket t \rrbracket_A (\Gamma^C \nu) \\
(A[\sigma])^I := A^I[\sigma^I]^I & (A[\sigma])^C \nu t := A^C (\sigma^I \circ \nu) t \\
\cdot^I := \cdot^I & \cdot^C \nu := \text{tt} \\
(\Gamma \triangleright A)^I := \Gamma^I \triangleright^I A^I & (\Gamma \triangleright A)^C \nu := (\Gamma^C (\pi_1 \nu), A^C (\pi_1 \nu) (\pi_2 \nu)) \\
U^I := U^I & U^C \nu a := \text{Tm}^I \Omega (\text{El}^I a) \\
(\text{El} a)^I := \text{El}^I a^I & (\text{El} a)^C \nu t := {}_{a^C \nu \#} t \\
(\Pi a B)^I := \Pi^I a^I B^I & (\Pi a B)^C \nu t := \lambda \alpha. B^C (\nu, {}_{a^C \nu \#} \alpha) (t \hat{\otimes}_{a^C \nu \#} \alpha) \\
(\hat{\Pi} T B)^I := \hat{\Pi}^I T B^I & (\hat{\Pi} T B)^C \nu t := \lambda \alpha. (B \alpha)^C \nu (t \hat{\otimes} \alpha)
\end{array}$$

► **Example 20.** Now, given a syntactic signature $\Omega : \text{Con}^I$, we get the constructors as an Ω -algebra by $\omega := (\llbracket \Omega \rrbracket_{\text{IC}_\Omega})^C \text{id}^I : \llbracket \Omega \rrbracket_A$. If Ω is the syntactic signature for natural numbers, we get the constructors as in Example 18.

An $a : \text{Tm}^I \Omega U$ is a sort term for the syntactic signature Ω . If Ω is the syntactic signature for natural numbers, a can only be N (1 as a de Bruijn index). If Ω is the syntactic signature for Con-Ty (Example 4), a can be Con , $\text{Ty} \text{ @ empty}$, $\text{Ty} \text{ @ (ext @ empty @ (U @ empty))}$, and so on. In any case, for such an a , we obtain $(\llbracket a \rrbracket_{\text{IC}_\Omega})^C \text{id}^I : \text{Tm}^I \Omega (\text{El} a) = \llbracket a \rrbracket_A \omega$. That is, the algebra interpretation of a sort term at the constructors is equal to terms of that sort.

A $t : \text{Tm}^I \Omega (\text{El} a)$ is a term of a sort type a constructed using the constructors in Ω . For natural numbers, such a t can only be s applied iteratively to z . For such a t , we obtain $(\llbracket t \rrbracket_{\text{IC}_\Omega})^C \text{id}^I : (t = \llbracket t \rrbracket_A \omega)$. That is, a constructor term is equal to its algebra interpretation at the constructors. This is exactly the equation needed at the end of Example 18.

► **Definition 21** (Eliminator signature algebra IE_-). Given an $\Omega : \text{Con}^I$, we use the abbreviation $\omega := \llbracket \Omega \rrbracket_{\text{IC}_\Omega} \text{id}^I$ as in Example 20. Assuming an $\omega^D : (\llbracket \Omega \rrbracket_{\text{ADS}})^D \omega$, we define the signature algebra IE_{ω^D} . It is equipped with a morphism $-^I : \text{SignMor} \text{IE}_{\omega^D} \mathbb{I}$. We define IE_{ω^D} by listing its components Con , Ty , Sub , and so on, one per row. Each component has two parts denoted by I and E . The I part just reuses the corresponding components of \mathbb{I} , thus the morphism $-^I$ is defined as the obvious projection. We omit the equality components, as they come from UIP or are trivial. We also omit the components for terms and substitutions as their E parts are uninformative equational reasonings.

$$\begin{array}{ll}
\Gamma^I : \text{Con}^I & \Gamma^E : (\nu : \text{Sub}^I \Omega \Gamma^I) \rightarrow \llbracket \Gamma \rrbracket^S (\llbracket \nu \rrbracket^A \omega) (\llbracket \nu \rrbracket^D \omega^D) \\
A^I : \text{Ty}^I \Gamma^I & A^E : (\nu : \text{Sub}^I \Omega \Gamma^I) (t : \text{Tm}^I \Omega (A^I[\nu])) \rightarrow \\
& \llbracket A \rrbracket^S (\Gamma^E \nu) (\llbracket t \rrbracket^A \omega) (\llbracket t \rrbracket^D \omega^D) \\
\sigma^I : \text{Sub}^I \Gamma^I \Delta^I & \sigma^E : \Delta^E (\sigma^I \circ \nu) = \llbracket \sigma \rrbracket^S (\Gamma^E \nu)
\end{array}$$

6:16 For Finitary Induction-Induction, Induction Is Enough

$$\begin{array}{ll}
t^! : \mathsf{Tm}^! \Gamma^! A^! & t^E : A^E \nu (t^! [\nu]) = \llbracket t \rrbracket^S (I^E \nu) \\
(A[\sigma])^! := A^! [\sigma^!] & (A[\sigma])^E \nu t := A^E (\sigma^! \circ \nu) t \\
\cdot^! := \cdot^! & \cdot^E \nu := \mathsf{tt} \\
(\Gamma \triangleright A)^! := \Gamma^! \triangleright^! A^! & (\Gamma \triangleright A)^E \nu := (I^E (\pi_1 \nu), A^E (\pi_1 \nu) (\pi_2 \nu)) \\
\mathsf{U}^! := \mathsf{U}^! & \mathsf{U}^E \nu a := \lambda \alpha. \llbracket \alpha \rrbracket^{\mathsf{C id}\#} (\llbracket \llbracket \alpha \rrbracket^{\mathsf{C id}\#} \alpha \rrbracket^{\mathsf{D}} \omega^{\mathsf{D}}) \\
(\mathsf{El} a)^! := \mathsf{El}^! a^! & (\mathsf{El} a)^E \nu t := (\llbracket a \rrbracket^S (I^E \nu) (\llbracket t \rrbracket^A \omega)) \stackrel{\llbracket t \rrbracket^{\mathsf{C id}}}{=} \llbracket a \rrbracket^S (I^E \nu) t \stackrel{\alpha^E \nu}{=} \llbracket t \rrbracket^{\mathsf{D}} \omega^{\mathsf{D}} \\
(\Pi a B)^! := \Pi^! a^! B^! & (\Pi a B)^E \nu t := \\
& \lambda \alpha. \llbracket \alpha \rrbracket^{\mathsf{C id}\#} (B^E (\nu, \llbracket \alpha \rrbracket^{\mathsf{C id}}, \llbracket \nu \rrbracket^{\mathsf{C id}\#} \alpha) (t \textcircled{\alpha} \llbracket a \rrbracket^{\mathsf{C id}}, \llbracket \nu \rrbracket^{\mathsf{C id}\#} u)) \\
(\hat{\Pi} T B)^! := \hat{\Pi}^! T B^! & (\hat{\Pi} T B)^E \nu t := \lambda \alpha. (B \alpha)^E \nu (t \hat{\alpha} \alpha)
\end{array}$$

► **Example 22.** Given the assumptions $\Omega, \omega^{\mathsf{D}}$ of IE , we obtain the eliminator by $\llbracket \Omega \rrbracket_{\mathsf{IE}_{\omega^{\mathsf{D}}}} \mathsf{id}^! : \llbracket \Omega \rrbracket^{\mathsf{S}} \omega \omega^{\mathsf{D}}$. The eliminator is a section of the displayed algebra ω^{D} , that is, a dependent function together with equalities witnessing that all the operations are preserved. If Ω is the syntactic signature for natural numbers, we get the eliminator of Example 18.

For a sort term $a : \mathsf{Tm}^! \Omega \mathsf{U}$, the interpretation $(\llbracket a \rrbracket_{\mathsf{IE}_{\omega^{\mathsf{D}}}})^E \mathsf{id}$ says that $(\lambda \alpha. \llbracket \alpha \rrbracket^{\mathsf{D}} \omega^{\mathsf{D}}) = \llbracket a \rrbracket^{\mathsf{S}} (\llbracket \Omega \rrbracket^E \mathsf{id})$, that is, the function for the sort a in the eliminator section is the displayed algebra interpretation at ω^{D} (motives and methods). For natural numbers, this is the same as $(\lambda n. \llbracket n \rrbracket^{\mathsf{D}} (N^{\mathsf{D}}, z^{\mathsf{D}}, s^{\mathsf{D}})) = (\lambda n. \mathsf{elimNat} (N^{\mathsf{D}}, z^{\mathsf{D}}, s^{\mathsf{D}}) n)$.

The interpretation of a constructor term $t : \mathsf{Tm}^! \Omega (\mathsf{El} a)$ is uninteresting as it provides an equality between two different equality proofs of the computation (β) rule for t .

► **Theorem 23.** *If a model of ETT supports the theory of IIT signatures, then it supports all IITs.*

Proof. For a signature $(\mathit{sig}, \mathit{sigc})$, we define constructors as

$$\mathit{con}_{\mathit{sig}} := (\llbracket \mathit{sig} \rrbracket_{\mathsf{IC}_{\mathit{sig}^!}})^{\mathsf{C id}^!} : (\mathit{sig} \mathsf{ADS})^{\mathsf{A}}$$

This typechecks as $\llbracket \mathit{sig} \rrbracket_{\mathsf{A}} = \llbracket - \rrbracket_{\mathsf{A}} (\mathit{sig} \mathsf{l}) \stackrel{\mathit{sigc}}{=} \mathit{sig} \mathsf{A} = (\mathit{sig} \mathsf{ADS})^{\mathsf{A}}$. We define the eliminator by and an

$$\mathit{elim}_{\mathit{sig}} \gamma^{\mathsf{D}} := (\llbracket \mathit{sig} \rrbracket_{\mathsf{IE}_{\gamma^{\mathsf{D}}}})^E \mathsf{id}^! : (\mathit{sig} \mathsf{ADS})^{\mathsf{S}} \mathit{con}_{\mathit{sig}} \gamma^{\mathsf{D}}.$$

This typechecks firstly because the type of γ^{D} matches the type of the parameter of IE :

$$(\mathit{sig} \mathsf{ADS})^{\mathsf{D}} \mathit{con}_{\mathit{sig}} \stackrel{\mathit{sigc}}{=} (\llbracket - \rrbracket_{\mathsf{ADS}} (\mathit{sig} \mathsf{l}))^{\mathsf{D}} \mathit{con}_{\mathit{sig}} = (\llbracket \mathit{sig} \rrbracket_{\mathsf{ADS}})^{\mathsf{D}} \mathit{con}_{\mathit{sig}},$$

and the result also has the correct type:

$$\llbracket \mathit{sig} \rrbracket^{\mathsf{S}} \mathit{con}_{\mathit{sig}} \gamma^{\mathsf{D}} = (\llbracket - \rrbracket_{\mathsf{ADS}} (\mathit{sig} \mathsf{l}))^{\mathsf{S}} \mathit{con}_{\mathit{sig}} \gamma^{\mathsf{D}} \stackrel{\mathit{sigc}}{=} (\mathit{sig} \mathsf{ADS})^{\mathsf{S}} \mathit{con}_{\mathit{sig}} \gamma^{\mathsf{D}}. \quad \blacktriangleleft$$

► **Remark 24.** In the above proof, we crucially relied on the sigc property to define the constructors (and the eliminator). This is why the simple Church encoding of signatures is not sufficient.

4 Constructing the Theory of IIT Signatures

In this section we show that any model of ETT which supports indexed W-types also supports the theory of signatures, and as a consequence of Theorem 23, all IITs. For this, we work in the internal language of a model of ETT supporting indexed W-types [3]. Indexed

W-types correspond to the usual notion of (possibly mutual) indexed inductive types. We use Agda-style notation to define such inductive families: we list the sorts and constructors and use pattern matching when eliminating from them. For an encoding of mutual inductive families as indexed W-types, see e.g. [23].

We construct the theory of IIT signatures in the following steps:

1. We view the theory of signatures as a type theory, and we define its untyped syntax as mutual inductive types together with typing judgments given by inductive relations on the untyped syntax. Then the syntax $\mathbb{I} : \text{SignAlg}$ is constructed using those untyped terms for which the typing relation holds.
2. We construct $\llbracket - \rrbracket : \text{SignMor} \mid M$ for arbitrary $M : \text{SignAlg}$, by:
 - a. defining a relation $- \sim -$ between the well-typed syntax and a given signature algebra. The idea is that given a syntactic context Γ and a semantic context Γ^M of the signature algebra M , we have $\Gamma \sim \Gamma^M$ if and only if $\llbracket \Gamma \rrbracket = \Gamma^M$, and similarly for types, terms, and substitutions;
 - b. showing that this relation is functional and thus obtaining a morphism.
3. Proving the uniqueness of this morphism by showing that any morphism $f : \text{SignMor} \mid M$ satisfies the relation. For example, for any syntactic context Γ we have $\Gamma \sim f \Gamma$.

The next sections detail each of these steps.

4.1 Syntax

The goal is to define the syntactic signature algebra where contexts are pairs of a precontext together with a well-formedness proof, and similarly for types, terms and substitutions.

Crucially, we do not have conversion relations for typed syntax, nor do we need to use quotients when constructing the syntax. This is possible because there are no β -rules in the theory of signatures. Hence, we consider only normal terms in the untyped syntax, and define weakening and substitution by recursion. Avoiding quotients is important for two reasons. First, it greatly simplifies formalisation. Second, we aim to reduce the theory of signatures only to inductive types, thus making Theorem 57 stronger.

Now we present the definition of the untyped syntax and the associated typing judgments.

4.1.1 Untyped Syntax and its Properties

► **Definition 25** (Untyped syntax). *The untyped syntax is defined as the following inductive datatype.*

(1) *Substitution calculus*

$\text{Con}^{\text{P}} : \text{Set}$

$\text{Ty}^{\text{P}} : \text{Set}$

$\text{Sub}^{\text{P}} : \text{Set}$

$\text{Tm}^{\text{P}} : \text{Set}$

$\cdot^{\text{P}} : \text{Con}^{\text{P}}$

$\epsilon^{\text{P}} : \text{Sub}^{\text{P}}$

$- \triangleright^{\text{P}} - : \text{Con}^{\text{P}} \rightarrow \text{Ty}^{\text{P}} \rightarrow \text{Con}^{\text{P}}$

$- ,^{\text{P}} - : \text{Sub}^{\text{P}} \rightarrow \text{Tm}^{\text{P}} \rightarrow \text{Sub}^{\text{P}}$

$\text{var}^{\text{P}} : \mathbb{N} \rightarrow \text{Tm}^{\text{P}}$

6:18 For Finitary Induction-Induction, Induction Is Enough

(2) *Universe*

$$U^P : \mathsf{Ty}^P$$

$$EI^P : \mathsf{Tm}^P \rightarrow \mathsf{Ty}^P$$

(3) *Inductive parameters*

$$\Pi^P : \mathsf{Tm}^P \rightarrow \mathsf{Ty}^P \rightarrow \mathsf{Ty}^P$$

$$- @^P - : \mathsf{Tm}^P \rightarrow \mathsf{Tm}^P \rightarrow \mathsf{Tm}^P$$

(4) *External parameters*

$$\hat{\Pi}^P : (T : \mathsf{Set}) \rightarrow (T \rightarrow \mathsf{Ty}^P) \rightarrow \mathsf{Ty}^P$$

$$\tilde{\Pi}^P : (T : \mathsf{Set}) \rightarrow (T \rightarrow \mathsf{Tm}^P) \rightarrow \mathsf{Tm}^P$$

$$- \hat{\alpha} - : \mathsf{Tm}^P \rightarrow (\alpha : T) \rightarrow \mathsf{Tm}^P$$

(5) *Default value*

$$\mathsf{err}^P : \mathsf{Tm}^P$$

Variables are modeled as de Bruijn indices, i.e. as natural numbers pointing to a position in the context. We use the additional default constructor $\mathsf{err}^P : \mathsf{Tm}^P$ in case of error (ill-scoped substitution). The typing judgments will not mention err^P . The main interest of err^P is that it behaves like a closed term (which the theory of signatures lacks), in the sense that it is invariant under substitution. This makes expected equalities about substitution true even in the ill-typed case, thus reducing the number of hypotheses for the corresponding lemmas (see Lemma 32).

We will define substitutions $-[-]$ of types and terms recursively.

Note that $(\Pi^P A B)[\sigma]$ should be defined as $\Pi^P (A[\sigma]) (B[\mathsf{wk}_0 \sigma, {}^P \mathsf{var}^P 0])$, and thus we need to define wk_0 , the weakening of substitutions. The basic idea is to increment the de Bruijn indices of all the variables. Actually, this is not so simple because of the Π^P type: we want to define $\mathsf{wk}_0 (\Pi^P A B)$ as the Π type of the weakening of A and B , but here, B must be weakened with respect to the second last variable of the context, rather than the last one. For this reason, we need to generalise the weakening as occurring anywhere in the context.

► **Definition 26** (Untyped weakening). *We define untyped weakening recursively on terms by the following functions.*

$$\mathsf{wk}_n : \mathsf{Ty}^P \rightarrow \mathsf{Ty}^P$$

$$\mathsf{wk}_n : \mathsf{Tm}^P \rightarrow \mathsf{Tm}^P$$

$$\mathsf{wk}_0 : \mathsf{Sub}^P \rightarrow \mathsf{Sub}^P$$

The natural number n specifies at which position of the context the weakening occurs. Here, wk_0 weakens with respect to the last variable.

Later, in Lemma 36, we show that weakening preserves typing. Stating a typing rule for this operation requires weakening at the middle of a context. This is why we define pairs of untyped contexts, which should be thought of as a splitting of a context at some position. We call the second context a telescope over the first one.

► **Definition 27** (Untyped telescopes). *An untyped telescope is given simply by a Con^P .*

► **Definition 28** (Merging of a context and a telescope).

$$-; - : \mathsf{Con}^P \rightarrow \mathsf{Con}^P \rightarrow \mathsf{Con}^P$$

$$\Gamma; \cdot := \Gamma$$

$$\Gamma; (\Delta \triangleright^P A) := (\Gamma; \Delta) \triangleright^P A$$

► **Definition 29** (Weakening for telescopes). *Weakening for telescopes is defined pointwise. $\|\Gamma\|$ denotes the length of the context Γ .*

$$\begin{aligned} \text{wk}_0 & : \text{Con}^P \rightarrow \text{Con}^P \\ \text{wk}_0 \cdot^P & := \cdot^P \\ \text{wk}_0 (\Delta \triangleright^P A) & := \text{wk}_0 \Delta \triangleright^P \text{wk}_{\|\Delta\|} A \end{aligned}$$

This will be used to give typing rules for telescopes in Definition 35.

► **Definition 30** (Untyped unary substitution). *We define single substitution by recursion on the presyntax:*

$$\begin{aligned} -[- := -] : \text{Ty}^P \rightarrow \mathbb{N} \rightarrow \text{Tm}^P \rightarrow \text{Ty}^P \\ -[- := -] : \text{Tm}^P \rightarrow \mathbb{N} \rightarrow \text{Tm}^P \rightarrow \text{Tm}^P \end{aligned}$$

This is enough to state the typing judgments: indeed, the typing rule for application involves only a unary substitution.

However, to construct the syntax as a signature algebra, we need to define parallel substitutions:

► **Definition 31** (Untyped substitution calculus).

$$\begin{aligned} -[-] : \text{Ty}^P \rightarrow \text{Sub}^P \rightarrow \text{Ty}^P \\ -[-] : \text{Tm}^P \rightarrow \text{Sub}^P \rightarrow \text{Tm}^P \\ - \circ - : \text{Sub}^P \rightarrow \text{Sub}^P \rightarrow \text{Sub}^P \end{aligned}$$

These can be defined either by iterating unary substitutions, or by recursion on untyped syntax: the two ways yield provably equal definitions. In the following, we assume that they are defined by recursion. We also make use of the following definition:

$$\begin{aligned} \text{keep} : \text{Sub}^P \rightarrow \text{Sub}^P \\ := \lambda \sigma. (\text{wk}_0 \sigma \cdot^P \text{var}^P 0) \end{aligned}$$

The idea is that if σ is a substitution from Γ to Δ , then $\text{keep } \sigma$ is a substitution between contexts $\Gamma \triangleright A[\sigma]$ and $\Delta \triangleright A$ for any type A where the last term is just a de Bruijn index 0. This occurs when defining $(\Pi^P A B)[\sigma]$ as $\Pi^P (A[\sigma]) (B[\text{keep } \sigma])$.

We define the identity substitution on a context Γ as follows, where $\text{keep}^{\|\Gamma\|}$ is keep iterated $\|\Gamma\|$ times:

$$\begin{aligned} \text{id}^P : \text{Con}^P \rightarrow \text{Sub}^P \\ := \lambda \Gamma. \text{keep}^{\|\Gamma\|} \epsilon^P \end{aligned}$$

► **Lemma 32** (Exchange laws for weakening and substitution). *Below, Z denotes either a term or a type and keep^n denotes the n times iteration of keep .*

$$\begin{aligned} \text{wk-wk} & : \text{wk}_{n+p+1}(\text{wk}_n Z) = \text{wk}_n(\text{wk}_{n+p} Z) \\ \text{wk}_n[n] & : (\text{wk}_n Z)[n := z] = Z \\ \text{wk}_+[] & : (\text{wk}_{n+p+1} Z)[n := \text{wk}_p u] = \text{wk}_{n+p}(Z[n := u]) \\ \text{wk}[+] & : (\text{wk}_n Z)[n + p + 1 := u] = \text{wk}_n(Z[n + p := u]) \\ [][+] & : Z[n := u][n + p := z] = Z[n + p + 1 := z][n := (u[p := z])] \end{aligned}$$

6:20 For Finitary Induction-Induction, Induction Is Enough

$$\begin{aligned} [\text{keep}^n\text{-wk}_0] & : Z[\text{keep}^n(\text{wk}_0 \sigma)] = \text{wk}_n(Z[\text{keep}^n \sigma]) \\ \text{wk}_n[\text{keep}^n\text{-},] & : (\text{wk}_n Z)[\text{keep}^n(\sigma, {}^P u)] = Z[\text{keep}^n \sigma] \\ [:=][\text{keep}] & : Z[n := u][\text{keep}^n \sigma] = Z[\text{keep}^{n+1} \sigma][n := u[\sigma]] \end{aligned}$$

Proof. By induction on the untyped syntax. ◀

► **Corollary 33.** *As particular cases for $n = 0$, we get*

$$\begin{aligned} \circ \text{wk}_0 & : \sigma \circ (\text{wk}_0 \tau) = \text{wk}_0(\sigma \circ \tau) \\ \text{wk}_0 \circ, & : \text{wk}_0 \sigma \circ (\tau, {}^P t) = \sigma \circ \tau \\ [\text{wk}_0] & : t[\text{wk}_0 \sigma] = \text{wk}_0(t[\sigma]) \\ \text{wk}_0[.,] & : (\text{wk}_0 Z)[\sigma, {}^P u] = Z[\sigma] \\ [0 :=][\text{keep}] & : Z[0 := u][\sigma] = Z[\text{keep} \sigma][0 := u[\sigma]] \end{aligned}$$

► **Lemma 34** (Composition functor law and associativity).

$$\begin{aligned} [\text{keep}] & : Z[\sigma][\tau] = Z[\sigma \circ \tau] \\ \text{ass} & : (\sigma \circ \delta) \circ \tau = \sigma \circ (\delta \circ \tau) \end{aligned}$$

We defer laws for identity substitutions after the definition of the typing judgments, as the proofs require that some inputs are well-typed.

4.1.2 Typing Relations and Their Properties

► **Definition 35** (Typing relations). *The typing relations are defined as the following inductive type indexed over the untyped syntax:*

(1) *Substitution calculus*

$$\begin{aligned} - \vdash & : \text{Con}^P \rightarrow \text{Set} \\ - \vdash - & : \text{Con}^P \rightarrow \text{Ty}^P \rightarrow \text{Set} \\ - \vdash - \in_{\mathbb{N}} - & : \text{Con}^P \rightarrow \mathbb{N} \rightarrow \text{Ty}^P \rightarrow \text{Set} \\ - \vdash - \in - & : \text{Con}^P \rightarrow \text{Tm}^P \rightarrow \text{Ty}^P \rightarrow \text{Set} \\ - \vdash - \Rightarrow - & : \text{Con}^P \rightarrow \text{Sub}^P \rightarrow \text{Con}^P \rightarrow \text{Set} \\ \cdot^w & : \cdot^P \vdash \\ \epsilon^w & : \Gamma \vdash \epsilon^P \Rightarrow \cdot^P \\ - \triangleright^w - & : (\Gamma \vdash) \rightarrow (\Gamma \vdash A) \rightarrow \Gamma \triangleright^P A \vdash \\ ,^w & : (\Delta \vdash) \rightarrow (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow (\Delta \vdash A) \rightarrow (\Gamma \vdash t \in A[\sigma]) \rightarrow \Gamma \vdash \sigma, {}^P t \Rightarrow \Delta \triangleright^P A \\ \text{var}^w & : (\Gamma \vdash n \in_{\mathbb{N}} A) \rightarrow \Gamma \vdash \text{var}^P n \in A \\ 0^w & : (\Gamma \vdash) \rightarrow (\Gamma \vdash A) \rightarrow \Gamma \triangleright^P A \vdash 0 \in_{\mathbb{N}} \text{wk}^P A \\ S^w & : (\Gamma \vdash) \rightarrow (\Gamma \vdash A) \rightarrow (\Gamma \vdash n \in_{\mathbb{N}} A) \rightarrow (\Gamma \vdash B) \rightarrow \Gamma \triangleright^P B \vdash S n \in_{\mathbb{N}} \text{wk}^P A \end{aligned}$$

(2) *Universe*

$$\begin{aligned} \mathbf{U}^w & : (\Gamma \vdash) \rightarrow \Gamma \vdash \mathbf{U}^P \\ \mathbf{El}^w & : (\Gamma \vdash) \rightarrow (\Gamma \vdash a \in \mathbf{U}^P) \rightarrow \Gamma \vdash \mathbf{El}^P a \end{aligned}$$

(3) *Inductive parameters*

$$\begin{aligned} \Pi^w & : (\Gamma \vdash) \rightarrow (\Gamma \vdash a \in \mathbf{U}^P) \rightarrow (\Gamma \triangleright^P \mathbf{El}^P a \vdash B) \rightarrow \Gamma \vdash \Pi^P a B \\ \text{app}^w & : (\Gamma \vdash) \rightarrow (\Gamma \vdash a \in \mathbf{U}^P) \rightarrow (\Gamma \triangleright^P \mathbf{El}^P a \vdash B) \\ & \rightarrow (\Gamma \vdash t \in \Pi^P a B) \rightarrow (\Gamma \vdash u \in \mathbf{El}^P a) \rightarrow \Gamma \vdash t \circledast^P u \in B[0 := u] \end{aligned}$$

(4) *External parameters*

$$\begin{aligned} \hat{\Pi}^w & : (T : \mathbf{Set}) \rightarrow (A : T \rightarrow \mathbf{Ty}^p) \rightarrow (\Gamma \vdash) \rightarrow ((t : T) \rightarrow \Gamma \vdash A t) \rightarrow \Gamma \vdash \hat{\Pi}^p T A \\ \text{app}^w & : (T : \mathbf{Set}) \rightarrow (A : T \rightarrow \mathbf{Ty}^p) \rightarrow (\Gamma \vdash) \rightarrow ((t : T) \rightarrow \Gamma \vdash A t) \\ & \rightarrow (\Gamma \vdash t \in \hat{\Pi}^p T A) \rightarrow (u : T) \rightarrow \Gamma \vdash t \hat{\otimes} u \in A u \end{aligned}$$

There is possibility of redundancy in the arguments of the constructors. Here, we are “paranoid” (nomenclature from [9]), so that we get more inductive hypotheses when performing recursion.

► **Lemma 36** (Weakening preserves typing).

$$\begin{aligned} \text{wk}_0^w & : (\Gamma \vdash A) \rightarrow (\Gamma; \Delta \vdash) \rightarrow \Gamma \triangleright^p A; \text{wk}_0 \Delta \vdash \\ \text{wk}^w & : (\Gamma \vdash A) \rightarrow (\Gamma; \Delta \vdash B) \rightarrow \Gamma \triangleright^p A; \text{wk}_0 \Delta \vdash \text{wk}_{\|\Delta\|} B \\ \text{wk}^w & : (\Gamma \vdash A) \rightarrow (\Gamma; \Delta \vdash t \in B) \rightarrow \Gamma \triangleright^p A; \text{wk}_0 \Delta \vdash \text{wk}_{\|\Delta\|} t \in \text{wk}_{\|\Delta\|} B \\ \text{wk}_0^w & : (\Gamma \vdash A) \rightarrow (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \Gamma \triangleright^p A \vdash \text{wk}_0 \sigma \Rightarrow \Delta \end{aligned}$$

Proof. By mutual induction on the typing relations. ◀

We show that judgments are stable under substitution.

► **Lemma 37** (Substitution preserves typing).

$$\begin{aligned} []^w & : (\Gamma \vdash) \rightarrow (\Delta \vdash A) \rightarrow (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \Gamma \vdash A[\sigma] \\ []^w & : (\Gamma \vdash) \rightarrow (\Delta \vdash t \in A) \rightarrow (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \Gamma \vdash t[\sigma] \in A[\sigma] \\ []^w & : (\Delta \vdash x \in_{\mathbb{N}} A) \rightarrow (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \Gamma \vdash x[\sigma] \in A[\sigma] \\ \circ^w & : (\Gamma \vdash) \rightarrow (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow (\Delta \vdash \tau \Rightarrow E) \rightarrow \Gamma \vdash \tau \circ \sigma \Rightarrow E \end{aligned}$$

Proof. By mutual induction on the typing relations. ◀

We show the category and functor laws involving identity substitution for well-formed types, terms and substitutions.

► **Lemma 38** (Identity laws).

$$\begin{aligned} [\text{id}^p] & : (\Gamma \vdash A) \rightarrow A[\text{id}^p \Gamma] = A \\ [\text{id}^p] & : (\Gamma \vdash x \in_{\mathbb{N}} A) \rightarrow x[\text{id}^p \Gamma] = Vx \\ [\text{id}^p] & : (\Gamma \vdash t \in A) \rightarrow t[\text{id}^p \Gamma] = t \\ \text{id}^p & : (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \sigma \circ \text{id}^p \Gamma = \sigma \\ \text{id}^p & : (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \text{id}^p \Delta \circ \sigma = \sigma \end{aligned}$$

Finally, we show that the identity substitution itself is well-typed:

► **Lemma 39** (Typing for the identity substitution).

$$\text{id}^w : (\Gamma \vdash) \rightarrow \Gamma \vdash \text{id}^p \Gamma \Rightarrow \Gamma$$

► **Definition 40** (Proposition). *A type is a proposition, or proof-irrelevant, if it has at most one inhabitant.*

$$\text{is-prop } T := (a : T) \rightarrow (a' : T) \rightarrow a = a'$$

6:22 For Finitary Induction-Induction, Induction Is Enough

► **Lemma 41** (Proof irrelevance of typing relations).

$$\begin{aligned} \text{Con}^{\text{wp}} &: \text{is-prop } (\Gamma \vdash) \\ \text{Ty}^{\text{wp}} &: \text{is-prop } (\Gamma \vdash A) \\ \text{Var}^{\text{wp}} &: \text{is-prop } (\Gamma \vdash x \in_{\mathbb{N}} A) \\ \text{Tm}^{\text{wp}} &: \text{is-prop } (\Gamma \vdash t \in A) \\ \text{Sub}^{\text{wp}} &: \text{is-prop } (\Gamma \vdash \sigma \Rightarrow \Delta) \end{aligned}$$

► **Lemma 42** (Unicity of typing).

$$\begin{aligned} \text{Tm}^{\text{w}} = \text{Ty} &: (\Gamma \vdash t \in A) \rightarrow (\Gamma \vdash t \in B) \rightarrow A = B \\ \text{Var}^{\text{w}} = \text{Ty} &: (\Gamma \vdash x \in_{\mathbb{N}} A) \rightarrow (\Gamma \vdash x \in_{\mathbb{N}} B) \rightarrow A = B \end{aligned}$$

Let us consider for instance the application constructor app^{w} : for a codomain type B it yields an overall type $C = B[0 := u]$ for an application. Even if C is known a priori, there may be another B for which $B[0 := u] = C$, possibly leading to many proofs that $t @^{\text{p}} u$ has type C . Unicity of typing solves this issue, as B is then uniquely determined by the type $\text{II}^{\text{p}} A B$ of t .

4.1.3 The Syntax as a Signature Algebra

► **Definition 43** (Syntax for the theory of signatures). *We define the syntax as an element of SignAlg by pairs of untyped syntax and typing relations:*

$$\begin{aligned} \text{Con}^{\text{l}} &:= \sum_{\Gamma} \Gamma \vdash \\ \text{Ty}^{\text{l}}(\Gamma, \Gamma^{\text{w}}) &:= \sum_A \Gamma \vdash A \\ \text{Tm}^{\text{l}}(\Gamma, \Gamma^{\text{w}})(A, A^{\text{w}}) &:= \sum_t \Gamma \vdash t \in A \\ \text{Sub}^{\text{l}}(\Gamma, \Gamma^{\text{w}})(\Delta, \Delta^{\text{w}}) &:= \sum_{\sigma} \Gamma \vdash \sigma \Rightarrow \Delta \end{aligned}$$

The other fields are given straightforwardly. Regarding the equations, it is enough to prove them only for the untyped syntactic part: as we argued in Lemma 41, the proofs of typing judgments are automatically equal.

► **Remark 44.** Up until Definition 43, UIP is not used. Function extensionality on the other hand is necessary because the untyped metatheoretic II takes a metatheoretic function as an argument. An example induction step that uses function extensionality is in Lemma 38, in particular in the case $(\hat{\text{II}} T A)[\text{id}] = \hat{\text{II}} T A$. Indeed, the left hand side of this equation is equal to $\hat{\text{II}} T (\lambda t. (A t)[\text{id}])$ by definition, whereas the induction hypothesis states that $(t : T) \rightarrow (A t)[\text{id}] = A t$.

4.2 Relating the Syntax to a Signature Algebra

It remains to show that the constructed syntax l is the initial signature algebra. To achieve this, we first define a relation between the syntax and any signature algebra, then show that the relation is functional, which lets us extract a signature morphism from the relation.

This approach is an alternative presentation of Streicher’s method for interpreting preterms in an arbitrary model of type theory [30]. Streicher first defines a family of partial maps from the presyntax to a model, then shows that the maps are total on well-formed input. We

have found that our approach is significantly easier to formalise. To see why, note that the right notion of partial map in type theory, which does not presume decidable definedness, is fairly heavyweight:

$$\text{PartialMap } A B := A \rightarrow ((P : \text{Set}) \times \text{is-prop } P \times (P \rightarrow B))$$

In the above definition, we notice an opportunity for converting a fibered definition of a type family into an indexed one; if we drop the propositionality for P for the time being, we may equivalently return a family indexed over B , which is exactly just a relation $A \rightarrow B \rightarrow \text{Set}$. Then, in our approach, we recover uniqueness of P through the functionality requirement on the $A \rightarrow B \rightarrow \text{Set}$ relation, and totality by already assuming well-formedness of A . In type theory, using indexed families instead of display maps is a common convenience, since the former are natively supported, while the latter require carrying around auxiliary propositional equalities.

4.2.1 The Functional Relation

Given an $M : \text{SignAlg}$, we define the functional relation satisfied by the $\llbracket - \rrbracket : \text{SignMor} \mid M$ by recursion on the typing judgments. If Γ is a context in \mathbb{I} and Γ^M is a semantic context (i.e. a context in the signature algebra M), we want to define a type $\Gamma \sim \Gamma^M$ equivalent to $\llbracket \Gamma \rrbracket = \Gamma^M$. Of course, at this stage, $\llbracket - \rrbracket$ is not available yet since the point of defining this relation is to construct $\llbracket - \rrbracket$ in the end.

For a type A in a context Γ , we want to define a relation $A \sim A^M$ that is equivalent to $\llbracket A \rrbracket = A^M$. For this equality to make sense, the semantic type A^M must live in the semantic context $\llbracket \Gamma \rrbracket$. But again, $\llbracket - \rrbracket$ is not yet available at this stage. Exploiting the expected equivalence between $\Gamma \sim \Gamma^M$ and $\llbracket \Gamma \rrbracket = \Gamma^M$, we may consider defining $A \sim A^M$ under the hypotheses that A^M lies in a semantics context Γ^M which is related to Γ . Then, the type of the relation for types is

$$(\Gamma : \text{Con}^{\mathbb{I}}) \rightarrow (A : \text{Ty}^{\mathbb{I}} \Gamma) \rightarrow (\Gamma^M : \text{Con}^M) \rightarrow (\Gamma \sim \Gamma^M) \rightarrow (A^M : \text{Ty}^M \Gamma^M) \rightarrow \text{Set}$$

Note that the relation on contexts must be defined mutually with the relation on types (see for example the case of context extension), but here, the relation on contexts appears as the type of an argument of the relation on types. We want to avoid using such recursive-recursive definitions as they are not allowed by the elimination principles of indexed inductive types, so we instead just remove the hypothesis $\Gamma \sim \Gamma^M$ from the list of arguments. We proceed similarly for terms and substitutions. Actually, this removal is not without harm. For example, consider relating the empty substitution $\Gamma \vdash \epsilon^P \Rightarrow \cdot^P$ to a semantic substitution $\sigma^M : \text{Sub}^M \Gamma^M \Delta^M$. We would like to assert that σ^M equals the empty semantic substitution ϵ^M , but this is not possible because typechecking requires that Δ^M is the empty semantic context. This is precisely what was ensured by the hypothesis $\cdot^{\mathbb{I}} \sim \Delta^M$ we removed. Our way out here is to state that σ^M is related to the empty substitution if the target semantic context Δ^M is empty, and, acknowledging this equality, if σ^M is the empty substitution.

Let us mention another possible solution for avoiding recursion-recursion: defining $A \sim A^M$ so that it is equivalent to $(e : \llbracket \Gamma \rrbracket = \Gamma^M) \times (\llbracket A \rrbracket =_{e\#} A^M)$. In comparison, our approach yields a more concise definition of the relation. For example, in the case of the universe, this would lead to the definition $\text{U}^w \Gamma^w \sim A^M := (\Gamma^w \sim \Gamma^M) \times (A^M = \text{U}^M)$, instead of our definition $\text{U}^w \Gamma^w \sim A^M := (A^M = \text{U}^M)$.

6:24 For Finitary Induction-Induction, Induction Is Enough

► **Definition 45** (Relation \sim). We define the relation by recursion on the typing judgments. In the following, we abbreviate $A^w \sim_{\Gamma^M} A^M$ by $A^w \sim A^M$ when Γ^M can be inferred, and similarly for terms and substitutions.

(1) *Substitution calculus*

$$\begin{aligned}
 - \sim - & : \Gamma \vdash \rightarrow \text{Con}^M \rightarrow \text{Set} \\
 - \sim_{\Gamma^M} - & : \Gamma \vdash A \rightarrow \text{Ty}^M \Gamma^M \rightarrow \text{Set} \\
 - \sim_{\Gamma^M \vdash A^M} - & : \Gamma \vdash t \in A \rightarrow \text{Tm}^M \Gamma^M A^M \rightarrow \text{Set} \\
 - \sim_{\Gamma^M \vdash A^M} - & : \Gamma \vdash x \in_{\mathbb{N}} A \rightarrow \text{Tm}^M \Gamma^M A^M \rightarrow \text{Set} \\
 - \sim_{\Gamma^M \Rightarrow \Delta^M} - & : \Gamma \vdash \sigma \Rightarrow \Delta \rightarrow \text{Sub}^M \Gamma^M \Delta^M \rightarrow \text{Set}
 \end{aligned}$$

$$\begin{aligned}
 \cdot^w \sim \Gamma^M & := \Gamma^M = \cdot^M \\
 \epsilon^w \sim_{\Gamma^M \Rightarrow E^M} \delta^M & := (e_E : E^M = \cdot^M) \times (\delta^M =_{e_E \#} \epsilon^M) \\
 (\Gamma^w \triangleright^w A^w) \sim \Delta^M & := \sum_{\Gamma^M} (\Gamma^w \sim \Gamma^M) \times \sum_{A^M} (A^w \sim A^M) \times \\
 & \quad (\Delta^M = \Gamma^M \triangleright^M A^M) \\
 (,^w \Delta^w \sigma^w A^w t^w) \sim_{\Gamma^M \Rightarrow E^M} \delta^M & := \sum_{\Delta^M} (\Delta^w \sim \Delta^M) \times \sum_{\sigma^M} (\sigma^w \sim \sigma^M) \times \\
 & \quad \sum_{A^M} (A^w \sim A^M) \times \sum_{t^M} (t^w \sim t^M) \times \\
 & \quad (e_E : E^M = \Delta^M \triangleright^M A^M) \times \\
 & \quad (\delta =_{e_E \#} \sigma^M, t^M) \\
 \text{var}^w x^w \sim t^M & := x^w \sim t^M \\
 0^w \Gamma^w A^w \sim_{\Delta^M \vdash B^M} t^M & := \sum_{\Gamma^M} (\Gamma^w \sim \Gamma^M) \times \sum_{A^M} (A^w \sim A^M) \times \\
 & \quad (e_{\Delta} : \Delta^M = \Gamma^M \triangleright^M A^M) \times \\
 & \quad (e_B : B^M =_{e_{\Delta} \#} \text{wk}^M A^M) \times (t^M =_{e_{\Delta}, e_B \#} \text{vz}^M) \\
 S^w \Gamma^w A^w n^w B^w \sim_{\Delta^M \vdash C^M} t^M & := \sum_{\Gamma^M} (\Gamma^w \sim \Gamma^M) \times \sum_{A^M} (A^w \sim A^M) \times \\
 & \quad \sum_{B^M} (B^w \sim B^M) \times \sum_{n^M} (n^w \sim n^M) \times \\
 & \quad (e_{\Delta} : \Delta^M = \Gamma^M \triangleright^M B^M) \times \\
 & \quad (e_C : C^M =_{e_{\Delta} \#} \text{wk}^M A^M) \times \\
 & \quad (t^M =_{e_{\Delta}, e_C \#} \text{vs}^M n^M)
 \end{aligned}$$

(2) *Universe*

$$\begin{aligned}
 \text{U}^w \Gamma^w A^w \sim A^M & := A^M = \text{U}^M \\
 \text{El}^w \Gamma^w a^w \sim A^M & := \sum_{a^M} (a^w \sim a^M) \times (A^M = \text{El}^M a^M)
 \end{aligned}$$

(3) *Inductive parameters*

$$\begin{aligned}
 \Pi^w \Gamma^w a^w B^w \sim C^M & := \sum_{a^M} (a^w \sim a^M) \times \sum_{B^M} (B^w \sim B^M) \\
 & \quad \times (C^M = \Pi^M a^M B^M)
 \end{aligned}$$

$$\begin{aligned} \text{app}^w \Gamma^w a^w B^w t^w u^w \sim_{\Gamma^M \vdash C^M} x^M &:= \sum_{a^M} (a^w \sim a^M) \times \sum_{B^M} (B^w \sim B^M) \times \\ &\quad \sum_{t^M} (t^w \sim t^M) \times \sum_{u^M} (u^w \sim u^M) \times \\ &\quad (e_C : C^M = B^M[0 := u^M]^M) \times \\ &\quad (x^M =_{e_C \#} t^M \hat{\otimes}^M u^M) \end{aligned}$$

(4) *Metatheoretic parameters*

$$\begin{aligned} \hat{\Pi}^w T A \Gamma^w A^w \sim B^M &:= \sum_{A^M} ((t : T) \rightarrow A^w \sim A^M t) \times (B^M = \hat{\Pi}^M T A^M) \\ \text{a}\hat{\text{p}}\text{p}^w T A \Gamma^w A^w t^w u \sim_{\Gamma^M \vdash B^M} x^M &:= \sum_{A^M} ((t : T) \rightarrow A^w \sim A^M t) \times \sum_{t^M} (t^w \sim t^M) \times \\ &\quad (e_B : B^M = \hat{\Pi}^M T A^M) \times (x^M =_{e_B \#} t^M \hat{\otimes}^M u) \end{aligned}$$

4.2.2 Right Uniqueness

Next, we prove that this relation is right unique. Then, we show that the relation is stable under weakening and substitution. The last step consists of showing left-totality, i.e. giving a related semantic counterpart to any well-typed context, type or term. Everything is proved by induction on the typing judgments.

► **Lemma 46** (Right uniqueness). *The relation is right unique in the following sense:*

$$\begin{aligned} \Sigma \sim^P : (\Gamma^w : \Gamma \vdash) &\rightarrow \text{is-prop} \left(\sum_{\Gamma^M} \Gamma^w \sim \Gamma^M \right) \\ \Sigma \sim^P : (A^w : \Gamma \vdash A) &\rightarrow \text{is-prop} \left(\sum_{A^M} A^w \sim A^M \right) \\ \Sigma \sim^P : (t^w : \Gamma \vdash t \in A) &\rightarrow \text{is-prop} \left(\sum_{t^M} t^w \sim t^M \right) \\ \Sigma \sim^P : (x^w : \Gamma \vdash x \in_{\mathbb{N}} A) &\rightarrow \text{is-prop} \left(\sum_{x^M} x^w \sim x^M \right) \\ \Sigma \sim^P : (\sigma^w : \Gamma \vdash \sigma \Rightarrow \Delta) &\rightarrow \text{is-prop} \left(\sum_{\sigma^M} \sigma^w \sim \sigma^M \right) \end{aligned}$$

► **Remark 47.** We mentioned that in order to avoid a recursive-recursive definition, we removed some hypotheses in the list of arguments of the relation. Such hypotheses are sometimes missed, for example in the case of the empty substitution or in the case of variables, requiring us to state additional equalities. Because of this, we need UIP to show that $\sum_{\Gamma^M} \Gamma \sim \Gamma^M$ and $\sum_{A^M} A \sim A^M$ are propositions. One may think that the use of UIP could be avoided by using the alternative verbose definition that we suggested before, expecting that $\sum_{\Gamma^M} \sum_{A^M} A \sim A^M$, rather than $\sum_{A^M} A \sim A^M$, is a proposition. However, this is not obvious. For example, we were not able to define $\text{El}^w \Gamma^w a^w \sim A^M$ in this fashion. In related work, Hugunin investigated constructing IITs without UIP [19] in cubical type theory, and demonstrated that well-formedness predicates used in syntactic algebras can subtly break in that setting. Also, while Hugunin does not use UIP, he only shows the simple version version of dependent elimination for the constructed IITs. Hence, the question whether IITs are reducible to inductive types in a UIP-free setting remains open.

4.2.3 Stability under Weakening and Substitution

Stability of the relation under weakening must be proved before stability under substitution. Indeed, in the proof of stability under substitution, the Π case requires to show that $\Pi(A[\sigma])(B[\text{keep } \sigma])$ is related to $\Pi^M(A^M[\sigma]^M)(B^M[\text{keep}^M \sigma]^M)$. We would like to apply the induction hypothesis, so we need to show that $\text{keep } \sigma = \text{wk}_0 \sigma, \text{p var}^P 0$ is related to $\text{keep}^M \sigma^M$, knowing that σ is related to σ^M . As $\text{keep } \sigma = \text{wk}_0 \sigma, \text{p var}^P 0$, we are left with showing that $\text{wk}_0 \sigma = \sigma \circ \text{wk}$ (where $\text{wk} = \text{wk}_0 \text{id}$) relates to its semantic counterpart.

To achieve that, we show that wk_0 preserves the relation, for types and terms. This requires to generalise a bit and show that wk_n preserves the relation, as $\text{wk}_0(\Pi A B) = \Pi(\text{wk}_0 A)(\text{wk}_1 B)$. But remember that wk_n performs a weakening in the middle of a context, so we first define the semantic counterpart of this:

$$\begin{aligned} \Sigma \text{wk}_0 \Rightarrow^M : (\Gamma^w : \Gamma \vdash) &\rightarrow (\Gamma^w \sim \Gamma^M) \rightarrow \\ &(\Delta^w : \Gamma; \Delta \vdash) \rightarrow (\Delta^w \sim \Delta^M) \rightarrow \\ &(A^M : \text{Ty}^M \Gamma^M) \rightarrow (\Delta'^M : \text{Con}^M) \times (\text{Sub}^M \Delta'^M \Delta^M) \end{aligned}$$

Here, Δ'^M should be thought of as the context Δ^M where the weakening has happened in the middle of the context, by inserting the type A^M after the prefix Γ^M . Indeed, we expect that Γ^M is a prefix of Δ^M , as Γ^M relates to Γ and Δ^M to $\Gamma; \Delta$. The substitution from the weakened context to the original one must be computed at the same time otherwise the induction hypothesis is not strong enough. Then, we separate the two components under the same (implicit) hypotheses:

$$\begin{aligned} \text{wk}_0^M A^M \Delta^M &: \text{Con}^M \\ \text{wk} \Rightarrow^M A^M \Delta^M &: \text{Sub}^M(\text{wk}_0^M A^M \Delta^M) \Delta^M \end{aligned}$$

Note that if recursion-recursion is available in the metatheory, wk_0^M and $\text{wk} \Rightarrow^M$ can be defined directly without introducing this intermediate $\Sigma \text{wk}_0 \Rightarrow^M$.

► **Lemma 48** (Weakening preserves typing). *The following statements are all under the hypotheses $(\Gamma^w : \Gamma \vdash)$, $(\Gamma^w \sim \Gamma^M)$, $(\Delta^w : \Gamma; \Delta \vdash)$, $(\Delta^w \sim \Delta^M)$, $(A^w : \Gamma \vdash A)$, and $(A^w \sim A^M)$.*

$$\begin{aligned} \text{wk}_0 \sim &: \text{wk}_0^w A^w \Delta^w \sim \text{wk}_0^M A^M \Delta^M \\ \text{wk} \sim &: (T^w : \Gamma; \Delta \vdash T) \rightarrow (T^w \sim T^M) \rightarrow \text{wk}^w A^w T^w \sim T^M[\text{wk}_0 \Rightarrow^M A^M \Delta^M]^M \\ \text{wk} \sim &: (t^w : \Gamma; \Delta \vdash t \in T) \rightarrow (t^w \sim t^M) \rightarrow \text{wk}^w A^w t^w \sim t^M[\text{wk}_0 \Rightarrow^M A^M \Delta^M]^M \\ \text{wk} \sim &: (x^w : \Gamma; \Delta \vdash t \in_{\mathbb{N}} T) \rightarrow (x^w \sim x^M) \rightarrow \text{wk}^w A^w x^w \sim x^M[\text{wk}_0 \Rightarrow^M A^M \Delta^M]^M \end{aligned}$$

Proof. By mutual induction on the typing judgments. ◀

► **Lemma 49** (Weakening of substitution preserves $- \sim -$).

$$\begin{aligned} \text{wk}_0 \sim &: (\Gamma^w : \Gamma \vdash) \rightarrow (\Gamma^w \sim \Gamma^M) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (A^w \sim A^M) \rightarrow \\ &(\sigma^w : \Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow (\sigma^w \sim \sigma^M) \rightarrow (\text{wk}_0^w A^w \sigma^w \sim \sigma^M \circ^M \text{wk}_0^M) \end{aligned}$$

Proof. By induction on the typing judgments. ◀

Next, we want to prove that given any well-typed substitution $\sigma : \text{Sub } \Gamma \Delta$ and semantic contexts Γ^M and Δ^M , related to Γ and Δ , respectively, there is a semantic substitution related to σ . In the extension case $\Gamma \vdash \sigma,^p t \Rightarrow \Delta \triangleright^p A$, the induction hypothesis provides σ^M, Δ^M, A^M related to their syntactic counterpart. However, the premises of the induction hypothesis for getting a relevant t^M require showing that the type $A^M[\sigma^M]^M$ is related to the syntactic type $A[\sigma]$.

► **Lemma 50** (Preservation of the relation by substitution for variables).

$$\llbracket \sim : (\sigma^w : \Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow (\sigma^w \sim \sigma^M) \rightarrow (x^w : \Delta \vdash x \in_{\mathbb{N}} A) \rightarrow (x^w \sim x^M) \rightarrow \llbracket^w x^w \sigma^w \sim x^M [\sigma^M]^M$$

Proof. Induction on typing. ◀

► **Lemma 51** (Preservation of the relation by substitution for types and terms). *We assume $(\sigma^w : \Gamma \vdash \sigma \Rightarrow \Delta)$, $(\sigma^w \sim \sigma^M)$, $(\Gamma^w : \Gamma \vdash)$, $(\Gamma^w \sim \Gamma^M)$, $(\Delta^w : \Delta \vdash)$, and $(\Delta^w \sim \Delta^M)$:*

$$\llbracket \sim : (A^w : \Delta \vdash A) \rightarrow (A^w \sim A^M) \rightarrow \llbracket^w \Gamma^w A^w \sigma^w \sim A^M [\sigma^M]^M$$

$$\llbracket \sim : (t^w : \Delta \vdash t \in A) \rightarrow (t^w \sim t^M) \rightarrow \llbracket^w \Gamma^w t^w \sigma^w \sim t^M [\sigma^M]^M$$

Proof. Mutual induction on typing. ◀

► **Lemma 52** (The relation is preserved by composition and identity). *We have the same hypotheses as in the previous lemma.*

$$\circ \sim : (E^w : E \vdash) \rightarrow (E^w \sim E^M) \rightarrow (\delta^w : \Delta \vdash \delta \Rightarrow E) \rightarrow (\delta^w \sim \delta^M) \rightarrow \circ^w \Gamma^w \delta^w \sigma^w \sim \delta^M \circ^M \sigma^M$$

$$\text{id} \sim : (\Gamma^w : \Gamma \vdash) \rightarrow (\Gamma^w \sim \Gamma^M) \rightarrow \text{id}^w \Gamma^w \sim \text{id}_{\Gamma^M}$$

4.2.4 Left-Totality and the Recursor

Before defining the recursor $\llbracket - \rrbracket$, we show left totality of the relation: that is, the image of a syntactic context is a unique semantic context which is related to it, and similarly for types and terms.

► **Lemma 53** (Left totality of $- \sim -$).

$$\Sigma \text{Con} \sim : (\Gamma^w : \Gamma \vdash) \rightarrow \sum_{\Gamma^M} \Gamma^w \sim \Gamma^M$$

$$\Sigma \text{Ty} \sim : (\Gamma^w : \Gamma \vdash) \rightarrow (\Gamma^w \sim \Gamma^M) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (A^M : \text{Ty}^M \Gamma^M) \times (A^w \sim A^M)$$

$$\Sigma \text{Tm} \sim : (\Gamma^w : \Gamma \vdash) \rightarrow (\Gamma^w \sim \Gamma^M) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (A^w \sim A^M) \rightarrow (t^w : \Gamma \vdash t \in A) \rightarrow (t^M : \text{Tm}^M \Gamma^M A^M) \times (t^w \sim t^M)$$

$$\Sigma \text{Var} \sim : (\Gamma^w : \Gamma \vdash) \rightarrow (\Gamma^w \sim \Gamma^M) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (A^w \sim A^M) \rightarrow (x^w : \Gamma \vdash x \in_{\mathbb{N}} A) \rightarrow (x^M : \text{Tm}^M \Gamma^M A^M) \times (x^w \sim x^M)$$

$$\Sigma \text{Sub} \sim : (\Gamma^w : \Gamma \vdash) \rightarrow (\Gamma^w \sim \Gamma^M) \rightarrow (\Delta^w : \Delta \vdash) \rightarrow (\Delta^w \sim \Delta^M) \rightarrow (\sigma^w : \Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow (\sigma^M : \text{Sub}^M \Gamma^M \Delta^M) \times (\sigma^w \sim \sigma^M)$$

Proof. By induction on well-formedness judgments. The right uniqueness of the relation is used in this induction. ◀

6:28 For Finitary Induction-Induction, Induction Is Enough

► **Lemma 54** (Existence of the recursor). *For any $M : \text{SignAlg}$ there is a $\llbracket - \rrbracket : \text{SignMor} \mid M$ where \mid is given in Definition 43.*

Proof. Using the first projections in the construction of the left-totally construction and right uniqueness. ◀

4.3 Uniqueness

It remains to show that the morphism constructed in Lemma 54 is unique. We exploit right uniqueness of the relation: it is enough to show that any such morphism maps a syntactic context to a related semantic context, and similarly for types and terms.

► **Lemma 55.** *We assume an arbitrary signature morphism f from \mid to M . This induces the following maps:*

$$\begin{aligned} \text{Con}^f &: (\Gamma \vdash) \rightarrow \text{Con}^M \\ \text{Ty}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (\Gamma \vdash A) \rightarrow \text{Ty}^M (\text{Con}^f \Gamma^w) \\ \text{Tm}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (\Gamma \vdash t \in A) \rightarrow \text{Tm}^M (\text{Con}^f \Gamma^w) (\text{Ty}^f \Gamma^w A^w) \\ \text{Var}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (\Gamma \vdash x \in_{\mathbb{N}} A) \rightarrow \text{Tm}^M (\text{Con}^f \Gamma^w) (\text{Ty}^f \Gamma^w A^w) \\ \text{Sub}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (\Delta^w : \Delta \vdash) \rightarrow (\Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \text{Sub}^M (\text{Con}^f \Gamma^w) (\text{Con}^f \Delta^w) \end{aligned}$$

The images of the above maps are related by $- \sim -$:

$$\begin{aligned} \sim \text{Con}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow \Gamma^w \sim \text{Con}^f \Gamma^w \\ \sim \text{Ty}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (A^w : \Gamma \vdash A) \rightarrow \Gamma^w \sim \text{Ty}^f \Gamma^w A^w \\ \sim \text{Tm}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (t^w : \Gamma \vdash t \in A) \rightarrow \Gamma^w \sim \text{Tm}^f \Gamma^w A^w t^w \\ \sim \text{Var}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (A^w : \Gamma \vdash A) \rightarrow (x^w : \Gamma \vdash x \in_{\mathbb{N}} A) \rightarrow \Gamma^w \sim \text{Var}^f \Gamma^w A^w x^w \\ \sim \text{Sub}^f &: (\Gamma^w : \Gamma \vdash) \rightarrow (\Delta^w : \Delta \vdash) \rightarrow (\sigma^w : \Gamma \vdash \sigma \Rightarrow \Delta) \rightarrow \Gamma^w \sim \text{Sub}^f \Gamma^w \Delta^w \sigma^w \end{aligned}$$

Proof. By induction on typing relations. ◀

► **Corollary 56** (Uniqueness of the recursor). *By right uniqueness of $- \sim -$, there is only one morphism $\text{SignMor} \mid M$ for any M .*

► **Theorem 57.** *If a model of ETT supports indexed W-types, it supports the theory of IIT signatures.*

Proof. We define the syntax \mid by Definition 43 which only used indexed W-types, the recursor by Lemma 54 and we prove its uniqueness property by Corollary 56. ◀

► **Corollary 58.** *If a model of ETT supports indexed W-types, it supports all IITs.*

Proof. Combining Theorem 57 and Theorem 23. ◀

5 Further Work

The current work only concerns finitary IITs. An extension would be to also allow infinitely branching inductive types such as W-types. This would first require giving semantics for infinitary IITs and adapting the term model construction. These would be straightforward following [24]. However, it seems to be more difficult to construct the syntax of infinitary

IIT signatures without using quotients. The reason is that such syntax would not be strictly restricted to neutral terms: the term model construction for infinitary IITs requires λ -abstraction and $\beta\eta$ -rules for infinitary Π types. A definition of normal preterms and typing judgments on them may still be possible, but it appears to be much more complicated than before (the current authors have attempted this without conclusive success).

As mentioned in Section 4.2.2, it also remains an open problem whether IITs are reducible to inductive types in a UIP-free setting. To show this, we would need to construct the syntax of signatures without UIP, and also reproduce the semantics and term model construction for IITs without UIP.

References

- 1 Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. From signatures to monads in unimath. *Journal of Automated Reasoning*, 63(2):285–318, August 2019. doi:10.1007/s10817-018-9474-4.
- 2 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham, 2018. Springer International Publishing.
- 3 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Program.*, 25, 2015. doi:10.1017/S095679681500009X.
- 4 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016. doi:10.1145/2837614.2837638.
- 5 Thorsten Altenkirch, Ambrus Kaposi, András Kovács, and Jakob von Raumer. Constructing inductive-inductive types via type erasure. In Marc Bezem, editor, *25th International Conference on Types for Proofs and Programs, TYPES 2019*. Centre for Advanced Study at the Norwegian Academy of Science and Letters, 2019.
- 6 Thorsten Altenkirch, Nuo Li, and Ondřej Rypáček. Some constructions on Ω -groupoids. In *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTTP '14*, pages 4:1–4:8, New York, NY, USA, 2014. ACM. doi:10.1145/2631172.2631176.
- 7 Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, page 503–515, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535852.
- 8 Steve Awodey, Jonas Frey, and Sam Speight. Impredicative encodings of (higher) inductive types. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 76–85. ACM, 2018. doi:10.1145/3209108.3209130.
- 9 Andrej Bauer, Philipp G. Haselwarter, and Théo Winterhalter. A modular formalization of type theory in Coq. In Ambrus Kaposi, editor, *23rd International Conference on Types for Proofs and Programs, TYPES 2017*. Eötvös Loránd University, 2017.
- 10 Guillaume Brunerie. A formalization of the initiality conjecture in agda. Slides of a talk at the Homotopy Type Theory 2019 Conference, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2019. URL: <https://guillaumebrunerie.github.io/pdf/initiality.pdf>.
- 11 Paolo Capriotti. Notions of type formers. In Ambrus Kaposi, editor, *23rd International Conference on Types for Proofs and Programs, TYPES 2017*. Eötvös Loránd University, 2017.

- 12 John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.
- 13 James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, January 2009. doi:10.1016/j.entcs.2008.12.114.
- 14 Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In Silvia Ghilezan and Ivetić Jelena, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016*. University of Novi Sad, 2016.
- 15 Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006. doi:10.1007/978-3-540-74464-1_7.
- 16 Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.
- 17 Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.
- 18 Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.
- 19 Jasper Hugunin. Constructing inductive-inductive types in cubical type theory. In Miłkołaj Bojańczyk and Alex Simpson, editors, *Foundations of Software Science and Computation Structures*, pages 295–312, Cham, 2019. Springer International Publishing.
- 20 Ambrus Kaposi. Re: separate definition of constructors? Email to the Agda mailing list, May 2019. URL: <https://lists.chalmers.se/pipermail/agda/2019/011176.html>.
- 21 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24, January 2019. doi:10.1145/3290315.
- 22 Ambrus Kaposi, András Kovács, and Ambroise Lafont. Closed inductive-inductive types are reducible to indexed inductive types. In Marc Bezem, editor, *25th International Conference on Types for Proofs and Programs, TYPES 2019*. Centre for Advanced Study at the Norwegian Academy of Science and Letters, 2019.
- 23 Ambrus Kaposi and Jakob von Raumer. A syntax for mutual inductive families. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, 2020. To appear.
- 24 András Kovács and Ambrus Kaposi. Large and infinitary quotient inductive-inductive types. In *35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2020, Saarbrücken, Germany, July 8-11, 2020*, 2020. To appear.
- 25 Lorenzo Malatesta, Thorsten Altenkirch, Neil Ghani, Peter Hancock, and Conor McBride. Small induction recursion, indexed containers and dependent polynomials are equivalent, 2013. TLCA 2013.
- 26 Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.
- 27 Fredrik Nordvall Forsberg and Anton Setzer. Inductive-inductive definitions. In Anuj Dawar and Helmut Veith, editors, *CSL 2010*, volume 6247 of *Lecture Notes in Computer Science*, pages 454–468. Springer, Heidelberg, 2010.
- 28 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
- 29 The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.
- 30 Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Birkhauser Boston Inc., Cambridge, MA, USA, 1991.

Eta-Equivalence in Core Dependent Haskell

Anastasiya Kravchuk-Kirilyuk

Princeton University, NJ, USA
ayk2@princeton.edu

Antoine Voizard

University of Pennsylvania, Philadelphia, PA, USA
voizard@seas.upenn.edu

Stephanie Weirich 

University of Pennsylvania, Philadelphia, PA, USA
sweirich@cis.upenn.edu

Abstract

We extend the core semantics for Dependent Haskell with rules for η -equivalence. This semantics is defined by two related calculi, Systems D and DC. The first is a Curry-style dependently-typed language with nontermination, irrelevant arguments, and equality abstraction. The second, inspired by the Glasgow Haskell Compiler's core language FC, is the explicitly-typed analogue of System D, suitable for implementation in GHC. Our work builds on and extends the existing metatheory for these systems developed using the Coq proof assistant.

2012 ACM Subject Classification Software and its engineering \rightarrow Functional languages; Software and its engineering \rightarrow Polymorphism; Theory of computation \rightarrow Type theory

Keywords and phrases Dependent types, Haskell, Irrelevance, Eta-equivalence

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.7

Supplementary Material <https://github.com/sweirich/corespec/tree/master>

Funding This material is based upon work supported by the National Science Foundation under Grant No. 1521539 and Grant No. 1704041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

In typed programming languages, the definition of type equality determines the expressiveness of the type system. If more types can (soundly) be shown to be equal, then more programs will type check. In dependently-typed languages, the definition of type equality relies on a definition of term equality, because terms may appear in types. Therefore, a dependently-typed language that can equate more terms can also admit more programs.

Many dependently-typed programming languages, such as Coq (since version 8.4) and Agda (from its initial design) include rules for η -equivalence when comparing functions for equality. These rules benefit programmers. For example, if a function f has type

$$f : P\ x \rightarrow \text{Int}$$

then it can be called with an argument of type

$$P\ (\lambda y. x\ y)$$

because the term $(\lambda y. x\ x)$ is η -equivalent to x .

Dependent Haskell [20, 47] is a proposal to add dependent types to the Haskell programming language, as implemented by the Glasgow Haskell Compiler. This design unifies the term and type languages of Haskell so that terms may appear directly in types, removing the need for awkward singleton encodings of richly-typed data structures [21, 27, 45].



© Anastasiya Kravchuk-Kirilyuk, Antoine Voizard, and Stephanie Weirich;
licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 7; pp. 7:1–7:31

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The specification of this language extension [47] is founded on two related dependently typed core calculi, called Systems D and DC. These two systems differ in their annotations: the latter language, which is inspired by and extends the FC intermediate language of GHC [42, 46], includes enough information to support simple, syntax-directed type checking. On the other hand, System D, is a Curry-style language meant to model the runtime behavior of the language, and to inspire type inference for the source language. (At the source level, type inference for Dependent Haskell will require more annotations than System D, which includes no annotations, and many fewer than System DC, which annotates everything.)

However, the specification of Systems D and DC, as presented in prior work, did not include rules for η -equivalence. The goal of this paper is to describe our experience with adding η -equivalence rules to these two systems, demonstrating that η -equivalence is compatible with Dependent Haskell.

While this extension is small—it involves three new rules for System D and two new rules for DC—it was not at all clear that it would work out from the beginning. Both Systems D and DC include support for *irrelevant arguments*, i.e the marking of some lambda-bound variables as not relevant for run-time execution. For Dependent Haskell, this feature is essential. Haskellers expect a type-erasure semantics and GHC erases type arguments during compilation. Irrelevance generalizes this idea to include not just type arguments but all terms that are used irrelevantly, enabling the generation of efficient code.

Unfortunately, η -equivalence, when combined with irrelevance in dependently-typed languages, is a subtle topic. Much prior work has laid out the issues, though in contexts that are not exactly the same as that found in Dependent Haskell. We describe this landscape in Section 6.3, and show how our work compares to and does not match any existing treatment of these features. In particular, our system features the `type:type` axiom, employs a typed definition of equivalence that ignores type annotations, supports large eliminations, includes a variant with decidable type checking, does not restrict how irrelevant arguments may be used in types, and comes with a completely mechanized type soundness proof.

In particular, this work extends the type soundness proof that was developed in prior work with support for η -equivalence. Prior work included a mechanized formalization of the meta-properties of both Systems D and DC, developed using the Coq proof assistant [43]. In this work, we have extended that development with these new rules and have updated the proofs accordingly. This mechanized proof gives us complete confidence in our extension, even in the face of a few curious findings.

As a result, this project also gives us a chance to report a success story for proof engineering. As the extension described in this paper is small compared to the overall system, we would expect the changes to the proof to be similarly minor, and they are. Furthermore, the three different forms of η -equivalence that we add are themselves quite similar to each other. Because of this relationship, a newcomer (the first author, an undergraduate at the time) could join the project and was able to adapt the changes needed for the usual η -equivalence rule to the novel ones for this setting. Although this process required careful understanding of binding representations, especially in the representation of the new rules, the mechanical proof served as an essential benefit to the overall research endeavor.

2 Overview of System D and System DC

This work presents and extends the languages Systems D and DC from prior work [47]. Therefore, we begin our discussion with an overview of these systems and their properties.

	D	DC
<i>Typing</i>	$\Gamma \vDash a : A$	$\Gamma \vdash a : A$
<i>Definitional equality (terms)</i>	$\Gamma; \Delta \vDash a \equiv b : A$	$\Gamma; \Delta \vdash \gamma : a \sim b$
<i>Proposition well-formedness</i>	$\Gamma \vDash \phi \text{ ok}$	$\Gamma \vdash \phi \text{ ok}$
<i>Definitional equality (props)</i>	$\Gamma; \Delta \vDash \phi_1 \equiv \phi_2$	$\Gamma; \Delta \vdash \gamma : \phi_1 \sim \phi_2$
<i>Context well-formedness</i>	$\vDash \Gamma$	$\vdash \Gamma$
<i>Signature well-formedness</i>	$\vDash \Sigma$	$\vdash \Sigma$
<i>Primitive reduction</i>	$\vDash a > b$	
<i>One-step reduction</i>	$\vDash a \rightsquigarrow b$	$\Gamma \vdash a \rightsquigarrow b$

■ **Figure 1** Summary of judgement forms.

System D is an implicit language; its syntax only contains terms that are relevant for computation. It is based on a Curry-style variant of a dependently-typed lambda calculus, with the `type:type` axiom. Functions are not annotated with their domain types and computations may not terminate. As a result, type checking in System D is undecidable. Compared to other Curry-style languages [32, 33], this language annotates the locations of irrelevant abstractions and irrelevant applications. Such generalizations and instantiations may occur only at the marked locations. Full Curry-style languages allow generalization and instantiation at any point in the derivation.

In contrast, System DC is an explicit language. It extends System D with enough annotations so that type checking is not only decidable, it is straightforward through a simple syntax-directed algorithm. While System D is intended to serve as a *specification* of what Dependent Haskell should mean, System DC is intended to serve as a core *implementation* language for the Glasgow Haskell Compiler (GHC) [20, 22], when it is extended with dependent types. The annotations allow the compiler to check core language terms during compilation, eliminating potential sources of bugs during compilation.

Because the annotated language DC is, in some sense, a *reification* of the derivations of D; DC can thus be seen as a syntax-directed version of D. To emphasize this connection in our formal system, we reuse the same metavariables for analogous syntactic forms in both languages.¹ The judgement forms are summarized in Figure 1. By convention, judgements for D use a double turnstile (\vDash) whereas judgements for DC use a single turnstile (\vdash). As we make precise below, judgements in these two languages are connected: we can apply an erasure operation to DC derivations to produce analogous judgements in D, and given a derivation in D, it is possible to add enough annotations to produce an analogous judgement in DC.

The judgement forms in these languages include the usual typing judgement, a typed equivalence relation (augmented in DC with an explicit proof witness in γ), a first-class notion of equality propositions ϕ , and a judgement when two propositions are equivalent (also augmented with a proof witness in DC), as well as well-formedness checks for typing contexts Γ and top-level signatures of recursive definitions Σ .

Computation in both languages is specified operationally, using a small-step, call-by-name, evaluation relation \rightsquigarrow . These one-step relations are decidable and produce a unique reduct in each case. This computation is also type sound, which we demonstrate through preservation and progress theorems [49].

¹ In fact, our Coq development uses the same syntax for both languages and relies on the judgement forms to identify the pertinent sets of constructs.

7:4 Eta-Equivalence in Core Dependent Haskell

System D

<i>terms, types</i>	a, b, A, B	$::=$	$\text{type} \mid x \mid F \mid \lambda^\rho x. b \mid a \ b^\rho \mid \square \mid \Pi^\rho x : A. B$ $\mid \Lambda c. a \mid a[\gamma] \mid \forall c : \phi. A$
<i>coercions</i>	γ	$::=$	\bullet
<i>values</i>	v	$::=$	$\lambda^+ x. a \mid \lambda^- x. v \mid \Lambda c. a$ $\mid \text{type} \mid \Pi^\rho x : A. B \mid \forall c : \phi. A$

System DC

<i>terms, types</i>	a, b, A, B	$::=$	$\text{type} \mid x \mid F \mid \lambda^\rho x : A. b \mid a \ b^\rho \mid \Pi^\rho x : A. B$ $\mid \Lambda c : \phi. a \mid a[\gamma] \mid \forall c : \phi. A$ $\mid a \triangleright \gamma$
<i>coercions (excerpt)</i>	γ	$::=$	$c \mid \text{refl } a \mid \text{sym } \gamma \mid \gamma_1; \gamma_2 \mid \text{red } a \ b \mid \dots$ $\text{eta } a$
<i>values</i>	v	$::=$	$\lambda^+ x : A. a \mid \lambda^- x : A. v \mid \Lambda c : \phi. a$ $\mid \text{type} \mid \Pi^\rho x : A. B \mid \forall c : \phi. A$

Shared syntax

<i>propositions</i>	ϕ	$::=$	$a \sim_A b$
<i>relevance</i>	ρ	$::=$	$+ \mid -$
<i>contexts</i>	Γ	$::=$	$\emptyset \mid \Gamma, x : A \mid \Gamma, c : \phi$
<i>available set</i>	Δ	$::=$	$\emptyset \mid \Delta, c$
<i>signature</i>	Σ	$::=$	$\emptyset \mid \Sigma \cup \{F \sim a : A\}$

■ **Figure 2** Syntax of D and DC. The syntactic differences between the two systems are highlighted in yellow. The sole addition for η -equivalence (the coercion form **eta** a) is highlighted in green.

The syntax of D, the implicit language, is shown at the top of Figure 2. This language, inspired by pure type systems [12], uses a shared syntax for terms and types. The language includes:

- a single sort (**type**) for classifying types,
- functions ($\lambda^+ x. a$) with dependent types ($\Pi^+ x : A. B$), and their associated application form ($a \ b^+$),
- functions with irrelevant arguments ($\lambda^- x. a$), their types ($\Pi^- x : A. B$), and instantiation form ($a \ \square^-$),
- coercion abstractions ($\Lambda c. a$), their types ($\forall c : \phi. B$), and instantiation form ($a[\bullet]$),
- and top-level recursive definitions (F).

In this syntax, term and type variables, x , are bound in the bodies of functions and their types. Similarly, coercion variables, c , are bound in the bodies of coercion abstractions and their types. (Technically, irrelevant variables and coercion variables are prevented by the

typing rules from actually appearing in the bodies of their respective abstractions.) We use the same syntax for relevant and irrelevant functions, marking which one we mean with a relevance annotation ρ . We sometimes omit relevance annotations ρ from applications $a\ b^\rho$ when they are clear from context. We also write nondependent relevant function types $\Pi^+ x:A.B$ as $A \rightarrow B$, when x does not appear free in B , and write nondependent coercion abstraction types $\forall c:\phi.A$ as $\phi \Rightarrow A$, when c does not appear free in A .

The metavariable Δ , called the *available set*, represents a set of coercion variables. This set is used to restrict the usage of coercion variables in certain situations; only variables appearing in the set are available.² The operation $\tilde{\Gamma}$ returns the available set made of all the coercion variables in the domain of context Γ . In other words, it is the available set that permits the use of all coercion variables in Γ .

The syntax of DC, also shown in the figure, includes the same features as D but with more typing annotations. In particular, this language removes the trivial argument for irrelevant instantiation (instead specifying the actual argument it stands for) and adds domain information to the bound variable in the abstraction forms. Finally, it replaces implicit type conversions by an explicit coercion term $a \triangleright \gamma$ as well as a language of coercion proofs (not completely shown in the figure). The addition of η -equivalence requires a new form of coercion proof, written **eta** a , that corresponds to all three new equivalence rules in D.

The erasure operation, written $|a|$ translates terms from System DC to System D by removing all type annotations and coercion proofs. For example, rules of this function include $|\lambda^\rho x:A.a| = \lambda^\rho x.|a|$ and $|a \triangleright \gamma| = |a|$.

2.1 Type checking in System D and System DC

Unlike System D, System DC enjoys unique typing, meaning that any given term has at most one type. Thanks to this uniqueness property and to the presence of typing annotations, type checking is decidable in System DC. In fact, the syntax of System DC can be seen as encoding not just a D term, but a D *typing derivation*. That is, any DC term uniquely identifies a typing derivation for the underlying (erased) D term.

In System D, type checking is undecidable due to two reasons. The first is that System D includes Curry-style System F as a sublanguage, where type checking is known to be undecidable [48, 36]. Since type arguments are implicit in Curry-style languages, irrelevant quantification is a feature of System D. The second reason for undecidable type checking in System D is the presence of an implicit conversion rule. In order to maintain decidable type checking in an environment where implicit conversion is allowed, System DC uses explicit coercion proofs whenever type conversion is performed. Below, we discuss these two features which contribute to the undecidability of type checking in System D. However, even though type checking is undecidable, we sketch what a partial type inference algorithm for System D might look like in Section 2.3.

2.1.1 Irrelevant quantification

Because Haskell includes parametric polymorphism, which has a type erasure semantics, Dependent Haskell includes a way to indicate which terms should be erased before execution.³ Thus, the rules that govern the treatment of irrelevant, or implicit, quantification appear in Figure 3.

² This is analogous to marking available coercion variables in the context.

³ Although it is possible to infer such information [14], we annotate it here to avoid a reliance on whole program optimization.

$$\begin{array}{c}
\text{E-PI} \\
\frac{\Gamma, x : A \vDash B : \text{type}}{\Gamma \vDash \Pi^\rho x : A.B : \text{type}} \\
\\
\text{E-ABS} \\
\frac{\Gamma, x : A \vDash a : B \quad (\rho = +) \vee (x \notin \text{fv } a)}{\Gamma \vDash \lambda^\rho x.a : \Pi^\rho x : A.B} \\
\\
\text{E-APP} \\
\frac{\Gamma \vDash b : \Pi^+ x : A.B \quad \Gamma \vDash a : A}{\Gamma \vDash b a^+ : B\{a/x\}} \\
\\
\text{E-IAPP} \\
\frac{\Gamma \vDash b : \Pi^- x : A.B \quad \Gamma \vDash a : A}{\Gamma \vDash b \square^- : B\{a/x\}}
\end{array}
\qquad
\begin{array}{c}
\text{AN-PI} \\
\frac{\Gamma, x : A \vdash B : \text{type}}{\Gamma \vdash \Pi^\rho x : A.B : \text{type}} \\
\\
\text{AN-ABS} \\
\frac{\Gamma, x : A \vdash a : B \quad (\rho = +) \vee (x \notin \text{fv } |a|)}{\Gamma \vdash \lambda^\rho x.A.a : \Pi^\rho x : A.B} \\
\\
\text{AN-APP} \\
\frac{\Gamma \vdash b : \Pi^\rho x : A.B \quad \Gamma \vdash a : A}{\Gamma \vdash b a^\rho : B\{a/x\}}
\end{array}$$

■ **Figure 3** Rules for relevant and irrelevant arguments in System D (left) and System DC (right).

D and DC's approach to implicit quantification follows ICC [32], ICC* [13], and EPTS [33]. When possible, the typing rules use the metavariable ρ to generalize over the relevance of the abstraction. For example, irrelevance places no restrictions on the usage of the bound variable in the body of the dependent function type, so the same rule suffices in each case (see rules E-PI and AN-PI).

However, for abstractions, if the argument is irrelevant, then the variable cannot appear in the body of the System D term (rule E-ABS). On the other hand, System DC includes annotations, which are not relevant, so the DC rule only restricts the variable from appearing in the *erasure* of the body (rule AN-ABS).

In DC, an application term is type-checked in the same way no matter whether it is relevant or not, so we are able to use the same rule in both cases (rule AN-APP). However, in D, if the application is to an irrelevant argument, then the argument does not appear in the term. Instead, it must be replaced by the trivial term \square (rule E-IAPP). Type-checking an irrelevant application in D thus requires guessing the actual argument used at this occurrence. Due to this, we need two separate rules for relevant and irrelevant application in D (rule E-APP and rule E-IAPP respectively).

2.1.2 Explicit coercions

As mentioned previously, System D includes an implicit conversion rule, shown on the left below (rule E-CONV). This rule depends on the type equality judgement to allow the system to work up-to the definition of this type equality. At any point in a System D derivation, the type of a term can silently be replaced with an equivalent type.

$$\begin{array}{c}
\text{E-CONV} \\
\frac{\Gamma \vDash a : A \quad \Gamma; \tilde{\Gamma} \vDash A \equiv B : \text{type}}{\Gamma \vDash a : B} \\
\\
\text{AN-CONV} \\
\frac{\Gamma \vdash a : A \quad \Gamma; \tilde{\Gamma} \vdash \gamma : A \sim B \quad \Gamma \vdash B : \text{type}}{\Gamma \vdash a \triangleright \gamma : B}
\end{array}$$

To enable decidable type checking, System DC includes an explicit justification γ in rule AN-CONV, called a coercion proof, whenever type conversion is used. These coercions are reifications of the type equality derivations of System D; a coercion proof γ specifies

a unique equality derivation. Equality is homogeneously typed in System D, if we have $\Gamma; \Delta \vDash a \equiv b : A$, then both terms a and b must have type A . In DC the relationship is more nuanced. If we have a coercion proof $\Gamma; \Delta \vdash \gamma : a \sim b$ where $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$, then there must exist an additional coercion proof witnessing the equality between types A and B . In other words, the types of coercible terms must be equal according to System D. For example, compare the reflexivity rule in System D below (rule E-REFL) with the two different reflexivity rules in System DC (rule AN-REFL and rule AN-ERASEEQ). While the first DC rule is the classic form of the reflexivity rule, we still need the second form to account for the case when two terms a and b have different type annotations. To derive reflexivity between a and b in this case, we must furthermore know that their *types* are equal, witnessed by the coercion proof γ . Note also that we cannot get away with having rule AN-ERASEEQ alone, since rule AN-REFL is the only rule which can derive reflexivity for type. For example, in order to prove $\mathbf{Int} \sim_{\text{type}} \mathbf{Int}$ with rule AN-ERASEEQ, we need the base case rule AN-REFL to prove $\text{type} \sim_{\text{type}} \text{type}$.

$$\begin{array}{c}
\text{E-REFL} \\
\frac{\Gamma \vDash a : A}{\Gamma; \Delta \vDash a \equiv a : A}
\end{array}
\qquad
\begin{array}{c}
\text{AN-REFL} \\
\frac{\Gamma \vdash a : A}{\Gamma; \Delta \vdash \mathbf{refl} \ a : a \sim a}
\end{array}
\qquad
\begin{array}{c}
\text{AN-ERASEEQ} \\
\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad |a| = |b| \quad \Gamma; \tilde{\Gamma} \vdash \gamma : A \sim B}{\Gamma; \Delta \vdash (a \mid_{\gamma} b) : a \sim b}
\end{array}$$

The type equality judgement in System D includes primitive (i.e. β) reductions, shown in rule E-BETA below. The analogous rule in System DC uses an explicit coercion, $\mathbf{red} \ a_1 \ a_2$ in the coercion checking rule AN-BETA to indicate a reduction. Both rules use the primitive reduction relation of System D, available in DC through erasure. Although this relation is deterministic, there are multiple ways to annotate a System D term. Thus, the coercion rule must annotate both terms, a_1 and a_2 involved in the redex. Furthermore, because these annotations may differ, these terms may have different types in DC, as long as those types are also related through erasure.

$$\begin{array}{c}
\text{E-BETA} \\
\frac{\Gamma \vDash a_1 : B \quad \vDash a_1 > a_2}{\Gamma; \Delta \vDash a_1 \equiv a_2 : B}
\end{array}
\qquad
\begin{array}{c}
\text{AN-BETA} \\
\frac{\Gamma \vdash a_1 : B_0 \quad \Gamma \vdash a_2 : B_1 \quad |B_0| = |B_1| \quad \vDash |a_1| > |a_2|}{\Gamma; \Delta \vdash \mathbf{red} \ a_1 \ a_2 : a_1 \sim a_2}
\end{array}$$

The System D type equality judgement is undecidable because it includes the operational semantics and the language is nonterminating. This nontermination is due to the `type:type` axiom and general recursion, the latter already available in Haskell. Furthermore, because System D is nonterminating, types themselves may diverge and thus don't necessarily have normal forms (this is already the case for GHC, in the presence of certain language extensions).

2.2 Coercion abstraction

D and DC inherit the *coercion abstraction* feature from System FC, the existing core language of GHC [42, 46]. This feature is primarily used to implement GADTs in GHC but is also available for explicit use by Haskell programmers.

Coercion abstraction means that equality is first class. Terms may abstract over equality propositions (denoted by ϕ in rules E-CABS and AN-CABS) and can discharge those assumptions in contexts where the proposition is derivable (rules E-CAPP and AN-CAPP). Once an equality has been assumed in the context, it may contribute to an equivalence derivation as long as the coercion variable is available (i.e. found in the available set Δ).

$$\begin{array}{c}
\text{E-CABS} \\
\frac{\Gamma, c : \phi \vDash a : B}{\Gamma \vDash \Lambda c. a : \forall c : \phi. B} \\
\\
\text{E-CAPP} \\
\frac{\Gamma \vDash a_1 : \forall c : (a \sim_A b). B_1 \quad \Gamma; \tilde{\Gamma} \vDash a \equiv b : A}{\Gamma \vDash a_1[\bullet] : B_1\{\bullet/c\}} \\
\\
\text{E-ASSN} \\
\frac{\vDash \Gamma \quad c : (a \sim_A b) \in \Gamma \quad c \in \Delta}{\Gamma; \Delta \vDash a \equiv b : A}
\end{array}
\qquad
\begin{array}{c}
\text{AN-CABS} \\
\frac{\Gamma, c : \phi \vdash a : B}{\Gamma \vdash \Lambda c : \phi. a : \forall c : \phi. B} \\
\\
\text{AN-CAPP} \\
\frac{\Gamma \vdash a_1 : \forall c : a \sim_{A_1} b. B \quad \Gamma; \tilde{\Gamma} \vdash \gamma : a \sim b}{\Gamma \vdash a_1[\gamma] : B\{\gamma/c\}} \\
\\
\text{AN-ASSN} \\
\frac{\vdash \Gamma \quad c : a \sim_A b \in \Gamma \quad c \in \Delta}{\Gamma; \Delta \vdash c : a \sim b}
\end{array}$$

The role of the set Δ is to prevent the usage of certain coercion variables, namely those introduced in a congruence proof between two coercion abstraction types. More details about this issue are available in prior work [47].

2.3 Type inference for System D

Even though complete type inference for System D is undecidable, we still intend it to be a model for the source language of the Glasgow Haskell Compiler. Type inference in GHC currently elaborates implicitly-typed Source Haskell to an explicitly-typed core language, similar to System DC. This inference algorithm works by gathering constraints and then solving those constraints using a variant of mixed-prefix unification combined with type-family reduction [44]. This algorithm already supports numerous features related to System D, including GADTs, type-level computation, higher-rank polymorphism and the `type:type` axiom. There are also experimental extensions of this algorithm in support of type-level lambdas [26], higher-kinds [50], and first-class polymorphism [39]. The most straightforward extension of GHC's algorithm with dependent types is based on parallel reduction; to determine whether two types are equivalent one must find a term that they both reduce to. In System D, this reduction may not terminate, so this process describes a semi-decision procedure.

3 Adding η -equivalence to Systems D and DC

Extending Systems D and DC with η -equivalence requires the addition of the following three rules to System D and two analogous rules in System DC. These three rules encode the usual η -equivalence properties for normal functions, irrelevant functions, and coercion abstractions. As our equivalence relation is typed, we must ensure that both left and right hand sides are well typed with the same type. This precondition also ensures that the bound variable does not appear free in b .

$$\begin{array}{c}
\text{E-ETAREL} \\
\frac{\Gamma \vDash b : \Pi^+ x : A. B}{\Gamma; \Delta \vDash \lambda^+ x. b \ x^+ \equiv b : \Pi^+ x : A. B} \\
\\
\text{E-ETAIRREL} \\
\frac{\Gamma \vDash b : \Pi^- x : A. B}{\Gamma; \Delta \vDash \lambda^- x. b \ \square^- \equiv b : \Pi^- x : A. B} \\
\\
\text{E-ETAC} \\
\frac{\Gamma \vDash b : \forall c : \phi. B}{\Gamma; \Delta \vDash \Lambda c. b[\bullet] \equiv b : \forall c : \phi. B}
\end{array}$$

In the annotated language, we only need two rules for coercion proofs because we can unify the two application forms in the annotated language (i.e. we can generalize over ρ).

$$\frac{\text{AN-ETA} \quad \Gamma \vdash b : \Pi^{\rho} x : A.B}{\Gamma; \Delta \vdash \mathbf{eta} \ b : \lambda^{\rho} x. b \ x^{\rho} \sim b} \quad \frac{\text{AN-ETAC} \quad \Gamma \vdash b : \forall c : \phi. B}{\Gamma; \Delta \vdash \mathbf{eta} \ b : \Lambda c. b[c] \sim b}$$

We use the single marker $\mathbf{eta} \ b$ as the explicit proof witness for both rules. We can overload this form because the annotated term b includes enough information to recover its type, and the type of b is enough to determine which of the η -equivalence properties are needed.

The five rules shown in this section are all that was needed to extend the definition of both languages with η -equivalence. Note that we do not include any η rules (i.e. reduction or expansion) in the operational semantics (i.e. the one step reduction relations $\vDash a \rightsquigarrow b$ and $\Gamma \vdash a \rightsquigarrow b$). The computational behavior of the system is unchanged by this extension. Instead, our goal is to extend the systems' reasoning about this existing computational behavior through the added equivalences. Although the rules for η -equivalence for relevant and irrelevant function have appeared in various prior work (see Section 6), the η -equivalence rule for coercion abstraction is new to this extension.

4 Extending proofs

The addition of the five rules above means that we must extend all existing proofs of Systems D and DC and show that after the inclusion of the new rules these systems retain the desired properties. The properties developed in prior work [47] include the following results.

- Consistency of definitional equality for System D
- Type soundness (progress and preservation) for both languages
- Decidable type checking for System DC
- Annotation and erasure lemmas relating the two languages

In this section, we provide an overview of these proofs and discuss their interaction with this extension. In the formal statements of our results below, we include the source file and definition in our Coq proofs⁴ that justifies that result.

The type soundness proof comes in two parts. We prove the progress lemma for System D, and then use the annotation lemma to translate that result to System DC. We prove the preservation lemmas for both systems directly, but it would also be possible to only prove preservation for System DC and then use the erasure lemma to translate that proof to System D.

By far, the largest modification was needed for the proof of the progress lemma for System D, which in turn relies on the consistency of definitional equality.

4.1 Progress lemma overview

In order to show proof of progress, we must first show the consistency of definitional equality in our setting (see Corollary 7 below). Consistency means that in certain contexts, types that have different head forms cannot be proven definitionally equal.

► **Definition 1** (Consistent⁵). *Two types A and B are consistent, written $\mathbf{consistent} \ A \ B$, when it is not the case that they are types with conflicting head forms. We formalize this property with the following two judgements.*

⁴ Available from <https://github.com/sweirich/corespec/tree/master/src/FcEtt>.

⁵ `ett.ott:consistent`

7:10 Eta-Equivalence in Core Dependent Haskell

<div style="border: 1px solid black; padding: 2px; display: inline-block;">$\text{hft}(A)$</div>	(Types with head forms)										
$\frac{\text{VALUE-TYPE-STAR}}{\text{hft}(\text{type})}$	$\frac{\text{VALUE-TYPE-PI}}{\text{hft}(\Pi^\rho x : A.B)}$	$\frac{\text{VALUE-TYPE-CPI}}{\text{hft}(\forall c : \phi.B)}$									
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 33%; text-align: center; vertical-align: top;"> <div style="border: 1px solid black; padding: 2px; display: inline-block;">$\text{consistent } a b$</div> </td> <td colspan="2" style="text-align: right; vertical-align: top;">(Types that do not differ in their heads)</td> </tr> <tr> <td style="text-align: center; vertical-align: top;"> $\frac{\text{CONSISTENT-A-STAR}}{\text{consistent type type}}$ </td> <td colspan="2" style="text-align: center; vertical-align: top;"> $\frac{\text{CONSISTENT-A-PI}}{\text{consistent } (\Pi^\rho x_1 : A_1.B_1) (\Pi^\rho x_2 : A_2.B_2)}$ </td> </tr> <tr> <td style="text-align: center; vertical-align: top;"> $\frac{\text{CONSISTENT-A-CPI}}{\text{consistent } (\forall c_1 : \phi_1.A_1) (\forall c_2 : \phi_2.A_2)}$ </td> <td style="text-align: center; vertical-align: top;"> $\frac{\text{CONSISTENT-A-STEP-R} \quad \neg(\text{hft}(b))}{\text{consistent } a b}$ </td> <td style="text-align: center; vertical-align: top;"> $\frac{\text{CONSISTENT-A-STEP-L} \quad \neg(\text{hft}(a))}{\text{consistent } a b}$ </td> </tr> </table>			<div style="border: 1px solid black; padding: 2px; display: inline-block;">$\text{consistent } a b$</div>	(Types that do not differ in their heads)		$\frac{\text{CONSISTENT-A-STAR}}{\text{consistent type type}}$	$\frac{\text{CONSISTENT-A-PI}}{\text{consistent } (\Pi^\rho x_1 : A_1.B_1) (\Pi^\rho x_2 : A_2.B_2)}$		$\frac{\text{CONSISTENT-A-CPI}}{\text{consistent } (\forall c_1 : \phi_1.A_1) (\forall c_2 : \phi_2.A_2)}$	$\frac{\text{CONSISTENT-A-STEP-R} \quad \neg(\text{hft}(b))}{\text{consistent } a b}$	$\frac{\text{CONSISTENT-A-STEP-L} \quad \neg(\text{hft}(a))}{\text{consistent } a b}$
<div style="border: 1px solid black; padding: 2px; display: inline-block;">$\text{consistent } a b$</div>	(Types that do not differ in their heads)										
$\frac{\text{CONSISTENT-A-STAR}}{\text{consistent type type}}$	$\frac{\text{CONSISTENT-A-PI}}{\text{consistent } (\Pi^\rho x_1 : A_1.B_1) (\Pi^\rho x_2 : A_2.B_2)}$										
$\frac{\text{CONSISTENT-A-CPI}}{\text{consistent } (\forall c_1 : \phi_1.A_1) (\forall c_2 : \phi_2.A_2)}$	$\frac{\text{CONSISTENT-A-STEP-R} \quad \neg(\text{hft}(b))}{\text{consistent } a b}$	$\frac{\text{CONSISTENT-A-STEP-L} \quad \neg(\text{hft}(a))}{\text{consistent } a b}$									

We use two auxiliary relations, *parallel reduction* and *joinability*, when proving consistency.

Parallel reduction, written $\vDash a \Rightarrow b$, is not part of the specification of System D⁶. This relation is a strongly confluent, but not necessarily terminating, rewrite relation on terms. In one step of parallel reduction, multiple redexes in one term may be reduced at the same time. For example, we can reduce $(z ((\lambda x.x) 1) ((\lambda y.y) 2))$ to $(z 1 2)$ in one step, even though two different beta-reductions need to be performed at the same time.

Two types are *joinable* when they reduce to some common term using any number of steps of parallel reduction.

► **Definition 2** (Joinable⁷). *Two types are joinable, written $\vdash a_1 \Leftrightarrow a_2$, when there exists some b such that $\vdash a_1 \Rightarrow^* b$ and $\vdash a_2 \Rightarrow^* b$.*

We use these two relations to prove consistency in two steps. First, we show that definitionally equal types are joinable. Second, we show that joinable types are consistent.

In proving the first step, it is important to note that only *some* definitionally equal types are joinable. This is illustrated by the following example. If a has type `type`, and there is a coercion assumption $a \sim_{\text{type}} \mathbf{Int}$ available in the context, then under this assumption a and `Int` are two definitionally equal types. However, these two types are not joinable. Because our consistency proof is based on parallel reduction, and because parallel reduction ignores assumed equality propositions, we state our result only for equality derivations with no available coercion assumptions. Thus, we restrict the set of all available assumptions we can use to derive equality to the empty set.

► **Theorem 3** (Equality implies Joinability⁸). *If $\Gamma; \emptyset \vDash a \equiv b : A$ then $\vdash a \Leftrightarrow b$*

This restriction in the lemma is necessary because the type system does not rule out clearly bogus assumptions, such as $\mathbf{Int} \sim_{\text{type}} \mathbf{Bool}$. Because we cannot use such assumptions to derive equality, they cannot be allowed to appear in the context. As a result, in order to be able to prove that consistent types are definitionally equal, the context must not make any such assumptions available.

To prove the second step, we use the fact that parallel reduction is a strongly confluent relation, and thus head forms must be preserved by parallel reduction. The confluence property is stated below.

⁶ `ett.ott:Par`

⁷ `ett.ott:join`

⁸ `ext_consist.v:consistent_defeq`

► **Theorem 4** (Confluence⁹). *If $\vDash a \Rightarrow a_1$ and $\vDash a \Rightarrow a_2$ then there exists b , such that $\vDash a_1 \Rightarrow b$ and $\vDash a_2 \Rightarrow b$.*

Our proof of confluence for System D follows the the proof of Church-Rosser for the untyped lambda calculus given in Barendregt [11], sections 3.2 and 3.2. The proof with β -reduction is attributed to Tait and Martin-Löf, and its extension with η -reduction is attributed to Hindley [25] and Rosen [38].

The confluence property essentially shows that even if a term can take several reduction paths, those paths can never diverge to produce terms with conflicting head forms. Thus, since joinability is defined in terms of parallel reduction, and parallel reduction is strongly confluent, it is true that joinability implies consistency.

► **Lemma 5** (Joinability is transitive¹⁰). *If $\vdash A_1 \Leftrightarrow B$ and $\vdash B \Leftrightarrow A_2$ then $\vdash A_1 \Leftrightarrow A_2$*

► **Theorem 6** (Joinability implies consistency¹¹). *If $\vdash A \Leftrightarrow B$ then **consistent** $A B$.*

► **Corollary 7** (Consistency). *If $\Gamma; \Delta \vDash a \equiv b : A$ then **consistent** $a b$.*

The consistency result allows us to prove the progress lemma for System D. This progress lemma is stated with respect to the one-step reduction relation and the definition of *value* given in Figure 2.

► **Lemma 8** (Progress¹²). *If $\Gamma \vDash a : A$, Γ contains no coercion assumptions, and no term variable x in the domain of Γ occurs free in a , then either a is a value or there exists some a' such that $\vDash a \rightsquigarrow a'$.*

4.2 Progress lemma update

The addition of η -equivalence required three new rules to be added to the parallel reduction relation. These rules encode η -reduction, meaning that any outer abstractions of the correct form can be removed. Because parallel reduction is an untyped relation, there is no analogous typing precondition as in the equivalence rules. However, these rules also have the condition that the bound variable not appear free in b or b' . (In our rules below, this condition is not explicitly mentioned because it is guaranteed by the usual Barendregt variable convention. We discuss how we maintain this property in our Coq development in Section 5.)

$$\frac{\text{PAR-ETA} \quad \vDash b \Rightarrow b'}{\vDash \lambda^+ x. b \ x^+ \Rightarrow b'} \quad \frac{\text{PAR-ETAIRREL} \quad \vDash b \Rightarrow b'}{\vDash \lambda^- x. b \ \square^- \Rightarrow b'} \quad \frac{\text{PAR-ETAC} \quad \vDash b \Rightarrow b'}{\vDash \Lambda c. b[\bullet] \Rightarrow b'}$$

We can view joinability as a semi-decision algorithm. Two terms are equal when they join to the same common reduct, though this process may diverge. This algorithm is a technical device only; we don't suggest its direct use in any implementation. Indeed, in the presence of η -reduction, joinability could equate more terms than definitional equality because it doesn't always preserve typing (see below).

⁹ `ext_consist.v:confluence`

¹⁰ `ext_consist.v:join_transitive`

¹¹ `ext_consist.v:join_consistent`

¹² `ext_consist.v:progress`

4.3 Parallel reduction and type preservation

There are three types of reduction included in this development: primitive reduction $\vDash a > b$, one-step reduction $\vDash a \rightsquigarrow b$, and parallel reduction $\vDash a \Rightarrow b$. In the original formulation of System D, all three of these reduction relations were type-preserving.

The first two relations are unchanged by this extension, so type preservation still holds for those relations¹³.

However, parallel reduction is an untyped relation. It does not depend on type information, even in the case of η -equivalence. As a result, after the addition of η -equivalence rules, the parallel reduction relation is no longer type-preserving.

► **Example 9** (Parallel reduction does not preserve types). There is some a such that $\Gamma \vDash a : A$ and $\vDash a \Rightarrow a'$ where there is no derivation of $\Gamma \vDash a' : A$.

This property fails in the case where $\lambda^+ x. b \ x^+$ reduces to b , but x is required in the context for b to type check, even though it does not appear free in b .

For example, let A be $\Pi^- x : \text{type}. \Pi^+ z : \text{type}. (x \rightarrow x)$ and consider the following derivation of the application of some function y with this type to two arguments: an implicit one and then an explicit one. In both cases in the derivation, the argument is just x , which is abstracted in the conclusion of the derivation.

$$\frac{\frac{\frac{\emptyset, y : A, x : \text{type} \vDash y : A \quad \emptyset, y : A, x : \text{type} \vDash x : \text{type}}{\emptyset, y : A, x : \text{type} \vDash y \ \square^- : \Pi^+ z : \text{type}. (x \rightarrow x)} \quad \emptyset, y : A, x : \text{type} \vDash x : \text{type}}{\emptyset, y : A, x : \text{type} \vDash y \ \square^- \ x^+ : x \rightarrow x}}{\emptyset, y : A \vDash \lambda^+ x. (y \ \square^-) \ x^+ : \Pi^+ x : \text{type}. (x \rightarrow x)}$$

Now, the term $\lambda^+ x. y \ \square^- \ x^+$ reduces to $y \ \square^-$ using rule PAR-ETA. However, there is no implicit argument that we can fill in so that this term will have type $\Pi^+ x : \text{type}. (x \rightarrow x)$.

Subject reduction also does not hold for η -reduction in the case of irrelevant arguments.¹⁴ In particular, there is a case where $\lambda^- x. b \ \square^-$ reduces to b and the two terms do not have the same type. This situation is not the same as above: the issue is that in a derivation of $\lambda^- x. b \ \square^-$ there is no requirement that the argument \square be the same type as x .

For example, suppose y has type $\Gamma \vdash y : \Pi^- x : A. B$ and we have $f : A \rightarrow A'$ in the context Γ where the type A does not equal A' . Then we can construct a derivation of $\Gamma \vdash \lambda^- x. (y \ \square^-) : \Pi^- x : A'. B\{f \ x/x\}$ by using the term $f \ x$ as the implicit argument. A similar counterexample also applies to η -reduction for coercion abstraction.

Thus, in the presence of η -reduction, preservation does not hold for parallel reduction. However, this loss is not significant to the soundness of the type systems of System D and System DC. None of our results require this property. The only place where this may come up is in a parallel-reduction based type inference algorithm for GHC (see Section 2.3). In this case, parallel reduction must preserve enough type information during reduction to ensure that the result is still well-typed.

4.4 Additional updates

Other updates to the proof include new cases in the erasure and annotation lemmas and in the uniqueness and decidability of type checking in DC. These lemmas are proven by mutual induction on the typing derivations shown in Figure 1. As the new rules are for the definitional/provable equality judgements, we only list that part of the lemma statement.

¹³ `ext_red.v:Beta_preservation, ext_red.v:reduction_preservation`

¹⁴ This issue was previously observed in the implementation of the Agda compiler: see <https://github.com/agda/agda/issues/2464>.

► **Lemma 10** (Erasure¹⁵). *If $\Gamma; \Delta \vdash \gamma : a \sim b$ then for all A such that $\Gamma \vdash a : A$, we have $|\Gamma|; \Delta \vDash |a| \equiv |b| : |A|$.*

► **Lemma 11** (Annotation¹⁶). *If $\Gamma; \Delta \vDash a \equiv b : A$ then for all Γ_0 , such that $|\Gamma_0| = \Gamma$, there exists some γ, a_0, b_0 and A_0 , such that $\Gamma_0; \Delta \vdash \gamma : a_0 \sim b_0$ and $\Gamma_0 \vdash a_0 : A_0$ and $\Gamma_0 \vdash b_0 : A_0$ where $|a_0| = a$ and $|b_0| = b$ and $|A_0| = A$.*

► **Lemma 12** (Unique typing for DC¹⁷). *If $\Gamma; \Delta \vdash \gamma : A_1 \sim B_1$ and $\Gamma; \Delta \vdash \gamma : A_2 \sim B_2$, then $A_1 = A_2$ and $B_1 = B_2$.*

► **Lemma 13** (Decidable typing for DC¹⁸). *Given Γ, Δ , and γ , it is decidable whether there exists some A and B such that $\Gamma; \Delta \vdash \gamma : A \sim B$.*

5 Proof engineering

The development of our Coq formalization for Systems D and DC was assisted with the use of two tools for mechanized reasoning about programming language metatheory. The first tool, Ott [40], takes as input a specification of the syntax and type system and produces both Coq definitions and LaTeX figures. The inference rules of this paper were typeset with this shared specification, though some rules in the main body of the paper have been slightly modified for clarity. We include the complete and unmodified specification of the system in Appendix A.

In addition to producing inductive definitions for the syntax and judgements, the Ott tool also produces substitution and free variable functions. To make working with these definitions more convenient, we also use the LNggen tool [9], that automatically states and proves many lemmas about these operations.

This extension increased the overall size of the original development by about ten percent, just looking at the line counts of the two versions. In Figure 4 we order the proof files by largest difference in line count¹⁹ to see that the most significant effort was the update to the progress proofs for System D. The preservation proof file (`ext_red.v`) shrank due to the removal of the preservation lemma for the parallel reduction relation. The table includes some modifications (such as inserting a newline, or slight refactoring of proof scripts) that have no effect on the development. Files with unchanged line counts are omitted from this figure.

The `ett_ind.v` file contains tactics that are tailored to our language development. These tactics automatically apply inference rules, pick fresh variables with respect to binders, etc. As we have added new rules to the language definition, we needed to update these tactics. To assist in the rest of this proof development, we developed a tactic for automatically rewriting a term given a hypothesis of the form found in the η -rules (and similar).

The `ext_invert.v` file contains inversion lemmas for System D. New with this extension is the addition of a lemma that asserts that \bullet is the only coercion proof found in System D terms.

¹⁵ `erase.v:typing_erase`

¹⁶ `erase.v:annotation_mutual`

¹⁷ `fc_unique.v:unique_mutual`

¹⁸ `fc_dec.v:FC_typechecking_decidable`

¹⁹ These numbers were calculated using the `cloc` tool, version 1.76, available from <http://github.com/AlDanial/cloc>.

7:14 Eta-Equivalence in Core Dependent Haskell

Specification (generated)	File name	(1)	(1 η)	(2)	(3)	(3 η)
	<code>ett_ott.v</code>	1337	1386	49	29	78
Progress (D)	<code>ext_consist.v</code>	1427	2054	627	205	832
Progress (D)	<code>ett_par.v</code>	660	1044	384	35	419
Erasure/annotation (D and DC)	<code>erase.v</code>	2002	2182	180	2	182
Decidability (DC)	<code>fc_dec_fun.v</code>	1561	1695	134	45	179
Progress (DC)	<code>fc_consist.v</code>	768	901	133	48	181
Inversion and regularity (D)	<code>ext_invert.v</code>	1057	1174	117	0	117
Inversion lemmas (DC)	<code>fc_invert.v</code>	650	665	15	82	97
Dec. of type checking (DC)	<code>fc_get.v</code>	774	844	70	1	71
General tactics	<code>ett_ind.v</code>	439	493	54	8	62
Preservation (D)	<code>ext_red.v</code>	290	241	-49	91	42
Context includes all vars (DC)	<code>fc_context_fv.v</code>	221	257	36	0	36
Context includes all vars (D)	<code>ext_context_fv.v</code>	143	178	35	0	35
Dec. of type checking (DC)	<code>fc_dec_aux.v</code>	395	399	4	18	22
Substitution (DC)	<code>fc_subst.v</code>	1270	1292	22	0	22
Unique typing (DC)	<code>fc_unique.v</code>	261	277	16	0	16
Reduction determinism (D)	<code>ext_red_one.v</code>	111	123	12	0	12
Substitution (D)	<code>ext_subst.v</code>	550	561	11	1	12
Primitive reduction	<code>beta.v</code>	71	78	7	4	11
Subst. prop. for coercions (DC)	<code>congruence.v</code>	349	354	5	0	5
Weakening (D)	<code>ext_weak.v</code>	139	141	2	3	5
Preservation (DC)	<code>fc_preservation.v</code>	247	245	-2	4	2
Well-formedness (D)	<code>ext_wf.v</code>	93	93	0	3	3
Dec. of type checking (DC)	<code>fc_dec_fuel.v</code>	223	223	0	2	2
Erasure properties	<code>erase_syntax.v</code>	486	486	0	1	1
General tactics	<code>tactics.v</code>	182	182	0	1	1
Total		17499	19404		554	2445

■ **Figure 4** Comparison between line counts in the original [47] and extended proof developments. The columns are (1) - number of lines in the original, (1 η) - number of lines in the extended version, (2) - change in line counts between the versions, (3) - size of diff for original, and (3 η) - size of diff for the extended version. Files that are identical between the versions are not included in the table, but appear in the total line count. Note, all line counts include only non-blank, non-comment lines of code.

5.1 Stating rules for η -equivalence

One issue that we faced in our development is the precise characterization of the new η -equivalence rules using Ott. In the end, our actual formalization specifies these rules in a slightly different form than as presented in Section 3. For example, rule PAR-ETA reads as follows, where we have named the body of the abstraction a and constrain it to be equal to the application as a premise of the rule.

$$\frac{\text{PAR-ETA} \quad \models b \Rightarrow b' \quad a = b x^+}{\models \lambda^+ x. a \Rightarrow b'}$$

Although informally, this is a minor change, the precise statement of the rule determines the definitions that will be produced in Coq.

The generated Coq definition uses the *locally nameless* representation and co-finite quantification [8] for the bound variable inside the abstraction. Given any choice for the bound variable x (except for some variables that must be avoided in the set L), we can show that *opening* the body of the abstraction²⁰ produces an application of b to that variable. Furthermore, because this equation must hold for almost any variable x , we know that x could not have appeared in the term b to begin with.

```

Inductive Par : context -> available_set -> tm -> tm -> Prop :=
...
| Par_Eta : forall (L:vars) (G:context) (D:available_set) (a b' b:tm),
  Par G D b b' ->
  (forall x, x \notin L ->
    open a (Var_f x) = App b Rel (Var_f x)) ->
  Par G D (UAbs Rel a) b'

```

In the Ott version of the rule, we need not explicitly mention that x cannot appear free in b due to this use of cofinite quantification. Thus, the usual side condition on η -reduction is implied by our formulation of the rule in Ott and does not need to be stated again.

5.2 Confluence proof update

Updating the confluence proof with the new cases for these rules was fairly straightforward. In particular, Coq was easily able to point out the new cases that needed to be added.

One wrinkle was that the new cases required a change from an induction on the syntax of the term to an induction on the *height* of the term. The reason for this modification is that the new η -rules reduce b , which is not an *immediate* subterm of $\lambda^+ x. b x^+$. However, it is clear that in comparison to $\lambda^+ x. b x^+$ the term b has a smaller height. The induction on height of term was also effective for the other cases where we were dealing with immediate subterms. Furthermore, our tool support (LNgen) already defined an appropriate height function for terms which we were able to use for this purpose. Consequently, although we needed to adjust the use of induction in each case, the overall modifications were minor.

²⁰The process of replacing the bound variable, represented by an index, with a free one.

6 Related work

6.1 Mechanized metatheory for dependent types

Mechanical reasoning via proof assistants has long been applied to dependent type theories. We will not attempt to describe all results. However, we will mention two recent developments:

Sozeau et al. [41] present the first implementation of a type checker for the kernel of Coq, which is proven correct in Coq with respect to its formal specification. More specifically, their work models an extension of the Predicative Calculus of (Co)-Inductive Constructions: a Pure Type System with an infinite hierarchy of universes, universe polymorphism, an impredicative sort, and inductive and co-inductive type families. However, although the Coq system includes η from version 8.4, this formalization does not include η -conversion. Like this work, their proofs of the metatheory of this system include a confluence proof of a parallel reduction relation, following Tait and Martin-Löf.

In [3], Abel, Öhman and Vezzosi mechanically prove (in Agda) the correctness of an algorithm for deciding conversion in a dependent type theory with one universe, an inductive type, and η -equality for function types. The algorithm that they verify is similar to the one used by Agda and is derived from Harper and Pfenning’s definition of LF [24], as refined and extended by Scherer and Abel [4, 2]. The proof of correctness of this algorithm is based on a Kripke logical relations argument, parameterized by suitable notion of equivalence of terms.

6.2 Dependent types, type:type and η -equivalence

Similarly, the literature is rich with work pertaining to η -equivalence in type theories. Below, we will focus on the interaction with type:type systems. In the next subsection, we discuss the interactions with irrelevant arguments.

Many versions of the type:type language do not include η -equivalence in the definition of conversion. For example, Coquand presents a semi-decision procedure for type checking a language with type:type [18]. This algorithm compares types for equality through weak-head normalization only. Similarly, Abel and Altenkirch [1] provide a more modern implementation of the type checking algorithm for a very similar language (still without η -conversion), and prove completeness on terminating terms (with a terminating type).

One difficulty with η -reduction in this setting is the problem with confluence for Church-style calculi. To avoid a dependency between type checking and reduction, many dependent type systems rely on an untyped reduction relation. However, in Church-style systems, parallel reduction is only confluent for well-typed terms; ill-typed terms may not have a common reduct. For example, the term $(\lambda x : A. (\lambda y : B. y) x)$ can η -convert to $\lambda y : B. y$ or β -convert to $\lambda x : A. x$. These terms are only equal when $A = B$, but that is only guaranteed by well-typed terms. As System D is a Curry-style system however, it does not suffer from this issue.

Two versions of type:type that include η -equivalence are Cardelli [15] and Coquand and Takeyama [19]. Both of these works justify the soundness of the type systems and the consistency of the conversion relation using a denotational semantics. Furthermore, in both of these systems, the denotational semantics ignores the annotated domain types of lambda-expressions.

Coquand and Takeyama additionally provide a semi-decidable type checking algorithm. Their conversion algorithm is not based on parallel reduction; instead it follows Coquand’s algorithm [17], reducing expressions to their weak-head-normal-forms before a structural comparison. When one of the terms being compared is a lambda expression and the other is not, the algorithm invents a fresh variable, applies both terms to this fresh variable and then continues checking for conversion.

	Q	DC	TE	η -F	η -T	Π	MM
P01 [37]	LF	✓	✓	✓	✓		
AS12 [4]	MLTT	✓	✓	✓	✓		
AVW17 [5]	MLTT	✓	✓	✓	✓	2.	
NVD17 [35]	MLTT	✓	✓			3.	
ND18 [34]	MLTT	4.	✓	✓	✓	3.	
A18 [7]	MLTT	4.	✓	5.	5.	✓	
M01 [32]	ECC			✓		✓	
BB08 [13]	ECC	✓		✓		✓	
MLS08 [33] IPTS	PTS					✓	
MLS08 [33] EPTS	PTS	✓				✓	
System D [47]	TT		✓	1.		✓	✓
System DC [47]	TT	✓	✓	1.		✓	✓

Notes:

1. Contribution of the current paper.
2. Only arguments of type *size* can be used without restriction.
3. Includes several different quantifiers, some with restriction, some without.
4. Not explicitly discussed in the paper. (But there are enough annotations that type checking is likely decidable.)
5. Definitional equality rules are not discussed in the paper, so the status is unclear.

■ **Figure 5** Dependent type systems with irrelevance.

6.3 Irrelevant quantification and η -equivalence

In this section, we survey prior work on dependently-typed languages that include some form of irrelevant quantification and discuss their interaction with η -equivalence. The contents of this section are summarized in Figure 5, which compares these systems along the features described below.

Note that the terms “irrelevance” and “irrelevant quantification” have multiple meanings in the literature. Our primary focus is on erasability, the ability for terms to quantify over arguments that need not be present at runtime. However, this terminology often includes compile-time irrelevance, or the blindness of type equality to such erasable parts of terms. It can also refer to erasability in the compile-time type equivalence algorithm. These terms are also related to, but not the same as, “parametricity” or “parametric quantification”, which characterizes functions that map equivalent arguments to equivalent results.

Below, we describe the various columns in this table that we use to lay out the design space of dependent type systems with irrelevance. Our purpose in this taxonomy is merely to define terms and summarize properties that we discuss below. We do not intend this table to characterize the contributions of prior work.

What form of type quantification is supported (Q)? First, we distinguish prior work by whether, and how, they support *type quantification*—that is, the ability for the system to quantify over types as well as terms. Type quantification is the foundation for *parametric polymorphism*, a key feature of modern programming languages, enabling modularity and code reuse. In dependent type systems, type quantification can take different forms, which have varying degrees of expressiveness. Prior work is based on the following foundations for type quantification:

LF [23], variants of the Logical Framework. This system includes dependency on terms only and does not allow quantification over types.

MLTT [30, 31], variants of Martin-Löf Type Theory. These systems feature predicative polymorphism only, where types are stratified into an infinite hierarchy of universes. A type from one universe can quantify only over types from lower universes.

ECC [16, 28], variants of the extended calculus of constructions. These systems feature an impredicative sort (called **Prop**), in addition to an infinite hierarchy of predicative universes. The types in the impredicative sort can quantify over themselves, all others must be stratified.

TT [29, 15], variants of core systems that include the `type:type` axiom. In these systems there is only a single sort of type, which includes types that quantify over all types. Systems D and DC include this form of quantification to make the system simpler for Haskell programmers, who are used to the impredicative polymorphism of System F.

PTS [10], pure type systems. These systems do not fix a single regime of type quantification. Instead, they may be instantiated with many different treatments of quantification, including all of the forms described above.

Is type checking decidable (DC)? Next, we distinguish systems based on whether they support decidable type checking (\checkmark) or not (\times). Some calculi include enough annotations so that a decidable type checking algorithm can be defined, others merely specify when terms are well-typed. Sometimes the “same” system can be cast in two different variants. For example, System D does not support decidable type checking, System DC augments the syntax of terms with annotations for this purpose.²¹

Is the definition of equality typed (TE)? Does the conversion rule in the type system use a typed (\checkmark) or untyped (\times) definition of equivalence? A typed equivalence requires a typed judgemental equality ([6]) and each transitive step used in the derivation to be between well-typed terms. In contrast, an untyped equivalence is usually defined in terms of β - or $\beta\eta$ -conversion of terms, only checking that the endpoints are well typed.

This distinction can affect expressiveness in both directions. On the one hand, an untyped relation might equate terms with different types, or justify an equality using ill-typed terms. There may be no analogous derivation in a typed relation. On the other hand, some equivalence rules (like η for the unit type, see below) can only be included in the system when type information is present, thus *expanding* the relation.

The inclusion of typed equivalence relation means that the algorithm used for type checking may depend not just on the syntax of terms but also on their types during execution. This type information may be used to prevent two terms from being equated (for example, if one of the terms doesn’t type check), or it may be used to enable two terms to be equated (such as in the case of the η -equivalence rule for the unit type).

Does the equality include η -equivalence rules for functions (η -F)? In this column, we include rules for functions regardless of whether they take relevant or irrelevant arguments. Note that some systems ([32]) do not mark the introduction and elimination sites of functions with irrelevant arguments. As a result, the corresponding equivalence rules are unnecessary. Similarly to other features, η -F (as well as η -T below) is important for programming as it may be used to derive equalities between types that mention functions, and thus to type-check more programs.

²¹Note, one typical location of annotation is the type of bound variables. Systems are often called “Church”-style when they include this annotation and “Curry”-style when they do not. However, this annotation is independent of the decidability of the type system, and many type systems that do not include this annotation support complete typing algorithms.

Does the equality include η -equivalence rules for products and unit (η -T)? Does the equality include type-directed η -equivalence rules for products or the unit type? For example, the rule for the unit type equates all terms of this type. Because this rule is type dependent, it can only be added to systems that use a typed definition of equivalence. These rules are typically implemented in the type system through a type-directed equivalence algorithm [24, 2].

At a high-level, the type-directed algorithm works in two stages. First, in the type-directed phase, if the terms being compared have function types, the two terms are applied to a fresh variable. This process takes care of η -equality. If the terms do not have function types, then the algorithm continues by converting both terms to weak-head normal form. If their heads match, then the algorithm recurses with the type-directed stage again on each of the corresponding subterms.

Is the codomain of the irrelevant Π -type unrestricted (Π)? In some systems, the *type* of an irrelevant abstraction is restricted so that the dependent argument must *also* be used irrelevantly. In other systems, the variable can appear freely without restrictions. Still others only allow unrestricted use for certain types of variables [5], or give users a choice [35, 34]. We discuss systems that include such restrictions, and their reasons for it, in Section 6.4. Systems D and DC do not restrict the codomain of irrelevant Π -types.

Mechanized metatheory (MM)? Have the metatheoretic results in the paper been developed and checked using a proof assistant? Our work is unique in this respect compared to similar systems.

6.4 Irrelevant quantification and restrictions on Π types

In this paper, we use irrelevance to mean erasure—i.e. the property that some arguments may be removed from the term without affecting the runtime behavior of the operational semantics. However, there is also a question of what happens to these arguments during type checking. Do these arguments affect the definition of type equality? If not, can they similarly be erased as part of a type checking algorithm?

Abel and Scherer [4] noted that although some arguments are irrelevant at run-time, they can still be relevant when determining type equality. If the definitional equality of the type system is typed, and if the type system allows *large eliminations*, i.e. the definition of a **type** via case analysis, then it can be difficult to incorporate type erasure into a type-directed equivalence algorithm. Fundamentally, the algorithm is driven by type information (instead of the structure of terms) and if irrelevant arguments can influence those types, they cannot be erased.

The key difficulty is demonstrated by the following example, taken from Abel and Scherer [4]. In the presence of large eliminations, and without any other restrictions, one would be able to type check the following term t , reproduced below in the syntax of DC extended with booleans.²²

²²Note that many systems support the large elimination needed for this example, even in the absence of inductive types. For example, in Systems D and DC we can use a Church-style encoding of booleans.

$$\begin{aligned}
T &: \mathbf{Bool} \rightarrow \text{type} \\
T &= \lambda^+ x: \mathbf{Bool}. \text{if } x \text{ then } (\mathbf{Bool} \rightarrow \mathbf{Bool}) \text{ else } \mathbf{Bool} \\
\\
t &= \lambda^- F: \Pi^- x: \mathbf{Bool}. (T x \rightarrow T x) \rightarrow \text{type}. \\
&\lambda^+ f: (F \mathbf{False}^- (\lambda^+ x: \mathbf{Bool}. x)^+) \rightarrow \mathbf{Bool}. \\
&\lambda^+ n: F \mathbf{True}^- (\lambda^+ x: (\mathbf{Bool} \rightarrow \mathbf{Bool}). \lambda^+ y: \mathbf{Bool}. x y^+)^+. \\
&\quad f (n \triangleright \gamma)^+
\end{aligned}$$

The DC coercion proof γ marks the point where conversion must be used in this example. This term is well-typed in a setting where the type system can derive an equality between the type of the parameter to f and the type of the argument n . These two types differ in only their irrelevant components, so they should be equated. In System DC, which, like ICC*, includes rules that erase types as part of type equivalence, we can define a coercion proof γ that witnesses the equality between the two types. Such a proof is composed transitively by first using the erasure-based reflexivity rule (rule AN-ERASEEQ) to change the implicit argument to F , and then using η -equivalence with the explicit argument.

$$\begin{aligned}
|F \mathbf{False}^- (\lambda^+ x: \mathbf{Bool}. x)^+| &= F \square^- (\lambda^+ x. x)^+ \\
&=_{\beta\eta} F \square^- (\lambda^+ x. \lambda^+ y. x y^+)^+ \\
&= |F \mathbf{True}^- (\lambda^+ x: (\mathbf{Bool} \rightarrow \mathbf{Bool}). \lambda^+ y: \mathbf{Bool}. x y^+)^+|
\end{aligned}$$

This example causes no difficulty for type checking in DC because it does not use a type-directed equivalence algorithm. Indeed, all of the information required by the algorithm is already present in the term.

However, it is difficult to extend a type-directed equivalence algorithm, particularly one that includes the η -equivalence rule for the unit type, so that it can equate these two types. Therefore, Abel and Scherer proposed restrictions on the use of irrelevantly quantified variables, not just in abstractions, but also in the codomain of irrelevant quantifiers. These restrictions were lifted in [5] for sized types, on the observation that they were irrelevant to the *shape* of types and therefore were not relevant to the operation of the type-equivalence algorithm. Nuyts and Devriese [35] expand on this idea and develop a modal type theory that includes, along with other modalities, irrelevance and shape-irrelevance in a unified framework.

However, note that the issue with this example is the desire to use erasure as part of a type-directed algorithm, not in the use of a typed equivalence in the language definition itself, nor the fact that the definition of type-equivalence ignores irrelevant components.

Because System DC does not rely on this sort of algorithm, it demonstrates that decidable type checking, irrelevance and large eliminations are compatible. Indeed, System DC requires the use of erasure in one of its key coercion proofs. On the other hand, one could worry that this example would cause trouble for System D. The fact that type checking is already undecidable in that language is not an excuse: a compiler like GHC will need to implement some type inference algorithm and should identify some subset of the language that it will support. This example demonstrates that type-directed algorithms are not a good fit for this setting, but does not rule out the algorithms sketched in Section 2.3.

7 Conclusion

Overall, this work demonstrates the benefits of developing the metatheory of type systems using a proof assistant. Although establishing the original development in prior work [47] took significant effort, we are able to build on that foundation when considering extensions of the system.

Furthermore, the availability of this sort of proof as a software engineering artifact makes it easier to bring on new collaborators. Because all of the proofs are machine-checked, newcomers can easily find what parts of the system need extension, even without understanding all details of how everything fits together. As a result, this sort of effort can be shared among many more collaborators, who can assist in maintaining the results.

Finally, the confidence gained from machine-checked proofs is also important. The failure of preservation for parallel η -reduction is obvious only in hindsight, and could have been easily overlooked in a pen-and-paper proof. At the same time, the automatic reassurance that this failure does not interact with the main soundness and decidability results is also welcome.

References

- 1 Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for Type:Type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). doi:10.1016/j.entcs.2011.02.013.
- 2 Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 3–12. IEEE, 2007.
- 3 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proceedings of the acm on programming languages*, 2(POPL):23, 2017.
- 4 Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1), 2012. doi:10.2168/LMCS-8(1:29)2012.
- 5 Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *PACMPL*, 1(ICFP):33:1–33:30, 2017. doi:10.1145/3110277.
- 6 ROBIN ADAMS. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, 2006. doi:10.1017/S0956796805005770.
- 7 Robert Atkey. The syntax and semantics of quantitative type theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*, 2018. doi:10.1145/3209108.3209189.
- 8 Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–15, January 2008.
- 9 Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, June 2010.
- 10 H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993.
- 11 Hendrik Pieter Barendregt. *The Lambda Calculus - its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.
- 12 Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

- 13 Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, pages 365–379, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 14 Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.
- 15 Luca Cardelli. A polymorphic λ -calculus with Type:Type. Technical report, DEC SRC, 1986. URL: <http://lucacardelli.name/Papers/TypeType.A4.pdf>.
- 16 Thierry Coquand. A calculus of constructions. manuscript, November 1986.
- 17 Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, New York, NY, USA, 1991.
- 18 Thierry Coquand. An algorithm for type-checking dependent types. *Science of computer programming.*, 26(1-3):167,177, 1996-05.
- 19 Thierry Coquand and Makoto Takeyama. An implementation of type: type. In *International Workshop on Types for Proofs and Programs*, pages 53–62. Springer, 2000.
- 20 Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016.
- 21 Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.
- 22 Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.
- 23 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. doi:10.1145/138027.138060.
- 24 Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Logic*, 6(1):61–101, January 2005. doi:10.1145/1042038.1042041.
- 25 J. Roger Hindley. *The Church-Rosser property and a result in combinatory logic*. PhD thesis, University of Newcastle upon Tyne, 1964.
- 26 Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. Higher-order type-level programming in haskell. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341706.
- 27 Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *ACM SIGPLAN Haskell Symposium*, 2013.
- 28 Z. Luo. ECC, an extended calculus of constructions. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 386–395, 1989.
- 29 Per Martin-Löf. A theory of types. Unpublished manuscript, 1971.
- 30 Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.
- 31 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.
- 32 Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.
- 33 Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008. doi:10.1007/978-3-540-78499-9_25.

- 34 Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 779–788. ACM, 2018. doi:10.1145/3209108.3209119.
- 35 Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *Proc. ACM Program. Lang.*, 1(ICFP):32:1–32:29, August 2017. doi:10.1145/3110276.
- 36 Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.
- 37 Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE Computer Society Press.
- 38 Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *J. ACM*, 20(1):160–187, January 1973. doi:10.1145/321738.321750.
- 39 Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. Guarded impredicative polymorphism. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 783–796. ACM, 2018. doi:10.1145/3192366.3192389.
- 40 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1), January 2010.
- 41 Matthieu Sozeau, Simon Boulter, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020. doi:10.1145/3371076.
- 42 M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007.
- 43 The Coq Development Team. The Coq proof assistant, version 8.8.0, April 2018. doi:10.5281/zenodo.1219885.
- 44 Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011. doi:10.1017/S0956796811000098.
- 45 Stephanie Weirich. Depending on types, 2014. Invited keynote given at ICFP 2014.
- 46 Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *Proceedings of The 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13*, pages 275–286, Boston, MA, September 2013.
- 47 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, August 2017. doi:10.1145/3110275.
- 48 J.B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111–156, 1999. doi:10.1016/S0168-0072(98)00047-5.
- 49 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994. doi:10.1006/inco.1994.1093.
- 50 Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. Kind inference for datatypes. *Proc. ACM Program. Lang.*, 4(POPL):53:1–53:28, 2020. doi:10.1145/3371121.

A Complete system specification

The complete type system appears in here including the actual rules that we used, automatically generated by Ott. For presentation purposes, we have removed some redundant hypotheses from these rules in the main body of the paper when they were implied via regularity. We have proven (in Coq) that these additional premises are admissible, so their removal does not change the type system.²³ These redundant hypotheses are marked by square brackets in the complete system below.

We need to include these redundant hypotheses in our rules for two reasons. First, sometimes these hypotheses simplify the reasoning and allow us to prove properties more independently of one another. For example, in the rule E-BETA rule, we require a_2 to have the same type as a_1 . However, this type system supports the preservation lemma so this typing premise will always be derivable. But, it is convenient to prove the regularity property early, so we include that hypothesis in the definition of the type system.

Another source of redundancy comes from our use of the Coq proof assistant. Some of our proofs require the use of induction on judgements that are not direct premises, but are derived from other premises via regularity. These derivations are always the same height or shorter than the original, so this use of induction is justified. However, while Coq natively supports proofs by induction on derivations, it does not natively support induction on the *heights* of derivations. Therefore, to make these induction hypotheses available for reasoning, we include them as additional premises.

Finally, instead of the usual syntactic distinction of values (as in Figure 2), our formalization identifies values using the judgement $[\text{Value } a]$, overloaded for both System D and System DC terms.

B Top-level signatures

Our results are proven with respect to the following top-level signatures:

$$\Sigma_1 = \emptyset \cup \{\mathbf{Fix} \sim \lambda^- x:\text{type}.\lambda^+ y:x.(y (\mathbf{Fix}[x] y)) : \Pi^- x:\text{type}.(x \rightarrow x) \rightarrow x\}$$

$$\Sigma_0 = |\Sigma_1|$$

However, our Coq proofs use these signature definitions opaquely. As a result, any pair of top-level signatures are compatible with the definition of the languages as long as they satisfy the following properties.

1. $\models \Sigma_0$
2. $\vdash \Sigma_1$
3. $\Sigma_0 = |\Sigma_1|$

²³ `ext_invert.v:E_Pi2,E_Abs2,E_CPi2,E_CAbs2,E_Fam2, ext_invert.v:E_Wff2,E_PiCong2,E_AbsCong2,E_CPiCong2,E_CAbsCong2, ext_red.v:E_Beta2, fc_invert.v:An_Pi_exists2,An_Abs_exists2,An_CPi_exists2,An_CAbs_exists2,An_Fam2, fc_invert.v:An_Sym2,An_Trans2,An_AbsCong_exists2,fc_invert.v:An_AppCong2,An_CPiCong_exists2,An_CAppCong2`

C Reduction relations

C.1 Primitive reduction

$$\boxed{\vDash a > b} \quad (\text{primitive reductions on erased terms})$$

$\frac{\text{BETA-APPABS}}{\vDash (\lambda^+ x.v) b^+ > v\{b/x\}}$	$\frac{\text{BETA-APPABSIRREL} \quad [\text{Value } (\lambda^- x.v)]}{\vDash (\lambda^- x.v) \square^- > v\{\square/x\}}$	$\frac{\text{BETA-CAPPCABS}}{\vDash (\Lambda c.a')[\bullet] > a'\{\bullet/c\}}$
$\frac{\text{BETA-AXIOM} \quad F \sim a : A \in \Sigma_0}{\vDash F > a}$		

C.2 System D one-step reduction

$$\boxed{\vDash a \rightsquigarrow b} \quad (\text{single-step head reduction for implicit language})$$

$\frac{\text{E-ABSTERM}}{\vDash a \rightsquigarrow a'}$	$\frac{\text{E-APPLEFT}}{\vDash a b^+ \rightsquigarrow a' b^+}$	$\frac{\text{E-APPLEFTIRREL}}{\vDash a \square^- \rightsquigarrow a' \square^-}$	$\frac{\text{E-CAPPLEFT}}{\vDash a[\bullet] \rightsquigarrow a'[\bullet]}$
$\frac{\text{E-APPABS}}{\vDash (\lambda^+ x.v) a^+ \rightsquigarrow v\{a/x\}}$	$\frac{\text{E-APPABSIRREL} \quad [\text{Value } (\lambda^- x.v)]}{\vDash (\lambda^- x.v) \square^- \rightsquigarrow v\{\square/x\}}$	$\frac{\text{E-CAPPCABS}}{\vDash (\Lambda c.b)[\bullet] \rightsquigarrow b\{\bullet/c\}}$	
$\frac{\text{E-AXIOM} \quad F \sim a : A \in \Sigma_0}{\vDash F \rightsquigarrow a}$			

C.3 System DC one-step reduction

$$\boxed{\Gamma \vdash a \rightsquigarrow b} \quad (\text{single-step, weak head reduction to values for annotated language})$$

$\frac{\text{AN-APPLEFT}}{\Gamma \vdash a \rightsquigarrow a'}$	$\frac{\text{AN-APPABS} \quad [\text{Value } (\lambda^\rho x:A.w)]}{\Gamma \vdash (\lambda^\rho x:A.w) a^\rho \rightsquigarrow w\{a/x\}}$	$\frac{\text{AN-CAPPLEFT}}{\Gamma \vdash a \rightsquigarrow a'}$
$\frac{\text{AN-APPABS}}{\Gamma \vdash a b^\rho \rightsquigarrow a' b^\rho}$	$\frac{\text{AN-ABSTERM} \quad \Gamma \vdash A : \text{type}}{\Gamma \vdash (\lambda^- x:A.b) \rightsquigarrow (\lambda^- x:A.b')}$	$\frac{\text{AN-AXIOM} \quad F \sim a : A \in \Sigma_1}{\Gamma \vdash F \rightsquigarrow a}$
$\frac{\text{AN-CAPPCABS}}{\Gamma \vdash (\Lambda c:\phi.b)[\gamma] \rightsquigarrow b\{\gamma/c\}}$	$\frac{\text{AN-CONVTERM} \quad \Gamma \vdash a \rightsquigarrow a'}{\Gamma \vdash a \triangleright \gamma \rightsquigarrow a' \triangleright \gamma}$	$\frac{\text{AN-COMBINE} \quad [\text{Value } v]}{\Gamma \vdash (v \triangleright \gamma_1) \triangleright \gamma_2 \rightsquigarrow v \triangleright (\gamma_1; \gamma_2)}$
$\frac{\text{AN-PUSH} \quad [\text{Value } v] \quad \Gamma; \tilde{\Gamma} \vdash \gamma : \Pi^\rho x_1:A_1.B_1 \sim \Pi^\rho x_2:A_2.B_2 \quad b' = b \triangleright \text{sym}(\text{piFst } \gamma) \quad \gamma' = \gamma @ (b' \mid_{(\text{piFst } \gamma)} b)}{\Gamma \vdash (v \triangleright \gamma) b^\rho \rightsquigarrow (v b'^\rho) \triangleright \gamma'}$	$\frac{\text{AN-CPUSH} \quad [\text{Value } v] \quad \Gamma; \tilde{\Gamma} \vdash \gamma : \forall c_1:\phi_1.A_1 \sim \forall c_2:\phi_2.A_2 \quad \gamma'_1 = \gamma_1 \triangleright \text{sym}(\text{cpiFst } \gamma) \quad \gamma' = \gamma @ (\gamma'_1 \sim \gamma_1)}{\Gamma \vdash (v \triangleright \gamma)[\gamma_1] \rightsquigarrow (v[\gamma'_1]) \triangleright \gamma'}$	

C.4 Parallel reduction

 $\boxed{\vDash a \Rightarrow b}$ *(parallel reduction (implicit language))*

$$\begin{array}{c}
\text{PAR-BETA} \\
\frac{\vDash a \Rightarrow (\lambda^+ x. a')}{\vDash b \Rightarrow b'} \\
\text{PAR-BETAIRREL} \\
\frac{\vDash a \Rightarrow (\lambda^- x. a')}{\vDash a \square^- \Rightarrow a' \{\square^- / x\}} \\
\text{PAR-APP} \\
\frac{\vDash a \Rightarrow a' \quad \vDash b \Rightarrow b'}{\vDash a b^+ \Rightarrow a' b'^+} \\
\text{PAR-REFL} \\
\frac{}{\vDash a \Rightarrow a} \\
\text{PAR-APPIRREL} \\
\frac{\vDash a \Rightarrow a'}{\vDash a \square^- \Rightarrow a' \square^-} \\
\text{PAR-CBETA} \\
\frac{\vDash a \Rightarrow (\Lambda c. a')}{\vDash a[\bullet] \Rightarrow a' \{\bullet / c\}} \\
\text{PAR-CAPP} \\
\frac{\vDash a \Rightarrow a'}{\vDash a[\bullet] \Rightarrow a'[\bullet]} \\
\text{PAR-ABS} \\
\frac{\vDash a \Rightarrow a'}{\vDash \lambda^\rho x. a \Rightarrow \lambda^\rho x. a'} \\
\text{PAR-PI} \\
\frac{\vDash A \Rightarrow A' \quad \vDash B \Rightarrow B'}{\vDash \Pi^\rho x : A. B \Rightarrow \Pi^\rho x : A'. B'} \\
\text{PAR-CABS} \\
\frac{\vDash a \Rightarrow a'}{\vDash \Lambda c. a \Rightarrow \Lambda c. a'} \\
\text{PAR-CPI} \\
\frac{\vDash A \Rightarrow A' \quad \vDash B \Rightarrow B' \quad \vDash a \Rightarrow a' \quad \vDash A_1 \Rightarrow A'_1}{\vDash \forall c : A \sim_{A_1} B. a \Rightarrow \forall c : A' \sim_{A'_1} B'. a'} \\
\text{PAR-AXIOM} \\
\frac{F \sim a : A \in \Sigma_0}{\vDash F \Rightarrow a} \\
\text{PAR-ETA} \\
\frac{\vDash b \Rightarrow b' \quad a = b x^+}{\vDash \lambda^+ x. a \Rightarrow b'} \\
\text{PAR-ETAIRREL} \\
\frac{\vDash b \Rightarrow b' \quad a = b \square^-}{\vDash \lambda^- x. a \Rightarrow b'} \\
\text{PAR-ETAC} \\
\frac{\vDash b \Rightarrow b' \quad a = b[\bullet]}{\vDash \Lambda c. a \Rightarrow b'}
\end{array}$$

D Full system specification: System D type system

 $\boxed{\Gamma \vDash a : A}$ *(typing)*

$$\begin{array}{c}
\text{E-STAR} \\
\frac{\vDash \Gamma}{\Gamma \vDash \text{type} : \text{type}} \\
\text{E-VAR} \\
\frac{\vDash \Gamma \quad x : A \in \Gamma}{\Gamma \vDash x : A} \\
\text{E-PI} \\
\frac{\Gamma, x : A \vDash B : \text{type} \quad [\Gamma \vDash A : \text{type}]}{\Gamma \vDash \Pi^\rho x : A. B : \text{type}} \\
\text{E-ABS} \\
\frac{\Gamma, x : A \vDash a : B \quad [\Gamma \vDash A : \text{type}] \quad (\rho = +) \vee (x \notin \text{fv } a)}{\Gamma \vDash \lambda^\rho x. a : \Pi^\rho x : A. B} \\
\text{E-APP} \\
\frac{\Gamma \vDash b : \Pi^+ x : A. B \quad \Gamma \vDash a : A}{\Gamma \vDash b a^+ : B\{a/x\}} \\
\text{E-IAPP} \\
\frac{\Gamma \vDash b : \Pi^- x : A. B \quad \Gamma \vDash a : A}{\Gamma \vDash b \square^- : B\{a/x\}} \\
\text{E-CONV} \\
\frac{\Gamma \vDash a : A \quad \Gamma; \tilde{\Gamma} \vDash A \equiv B : \text{type} \quad [\Gamma \vDash B : \text{type}]}{\Gamma \vDash a : B} \\
\text{E-CPI} \\
\frac{\Gamma, c : \phi \vDash B : \text{type} \quad [\Gamma \vDash \phi \text{ ok}]}{\Gamma \vDash \forall c : \phi. B : \text{type}} \\
\text{E-CABS} \\
\frac{\Gamma, c : \phi \vDash a : B \quad [\Gamma \vDash \phi \text{ ok}]}{\Gamma \vDash \Lambda c. a : \forall c : \phi. B} \\
\text{E-CAPP} \\
\frac{\Gamma \vDash a_1 : \forall c : (a \sim_A b). B_1 \quad \Gamma; \tilde{\Gamma} \vDash a \equiv b : A}{\Gamma \vDash a_1[\bullet] : B_1\{\bullet/c\}} \\
\text{E-FAM} \\
\frac{\vDash \Gamma \quad F \sim a : A \in \Sigma_0 \quad [\emptyset \vDash A : \text{type}]}{\Gamma \vDash F : A}
\end{array}$$

 $\boxed{\Gamma \vDash \phi \text{ ok}}$ *(Prop wellformedness)*

$$\begin{array}{c}
\text{E-WFF} \\
\frac{\Gamma \vDash a : A \quad \Gamma \vDash b : A \quad [\Gamma \vDash A : \text{type}]}{\Gamma \vDash a \sim_A b \text{ ok}}
\end{array}$$

$$\boxed{\Gamma; \Delta \vDash \phi_1 \equiv \phi_2}$$

(prop equality)

$$\begin{array}{c} \text{E-PROP CONG} \\ \Gamma; \Delta \vDash A_1 \equiv A_2 : A \\ \Gamma; \Delta \vDash B_1 \equiv B_2 : A \\ \hline \Gamma; \Delta \vDash A_1 \sim_A B_1 \equiv A_2 \sim_A B_2 \end{array} \quad \begin{array}{c} \text{E-ISO CONV} \\ \Gamma; \Delta \vDash A \equiv B : \text{type} \\ \Gamma \vDash A_1 \sim_A A_2 \text{ ok} \\ \Gamma \vDash A_1 \sim_B A_2 \text{ ok} \\ \hline \Gamma; \Delta \vDash A_1 \sim_A A_2 \equiv A_1 \sim_B A_2 \end{array}$$

$$\begin{array}{c} \text{E-CPiFST} \\ \Gamma; \Delta \vDash \forall c: \phi_1. B_1 \equiv \forall c: \phi_2. B_2 : \text{type} \\ \hline \Gamma; \Delta \vDash \phi_1 \equiv \phi_2 \end{array}$$

$$\boxed{\Gamma; \Delta \vDash a \equiv b : A}$$

(definitional equality)

$$\begin{array}{c} \text{E-ASSN} \\ \vDash \Gamma \quad c : (a \sim_A b) \in \Gamma \\ c \in \Delta \\ \hline \Gamma; \Delta \vDash a \equiv b : A \end{array} \quad \begin{array}{c} \text{E-REFL} \\ \Gamma \vDash a : A \\ \hline \Gamma; \Delta \vDash a \equiv a : A \end{array} \quad \begin{array}{c} \text{E-SYM} \\ \Gamma; \Delta \vDash b \equiv a : A \\ \hline \Gamma; \Delta \vDash a \equiv b : A \end{array} \quad \begin{array}{c} \text{E-TRANS} \\ \Gamma; \Delta \vDash a \equiv a_1 : A \\ \Gamma; \Delta \vDash a_1 \equiv b : A \\ \hline \Gamma; \Delta \vDash a \equiv b : A \end{array}$$

$$\begin{array}{c} \text{E-BETA} \\ \Gamma \vDash a_1 : B \\ [\Gamma \vDash a_2 : B] \quad \vDash a_1 > a_2 \\ \hline \Gamma; \Delta \vDash a_1 \equiv a_2 : B \end{array} \quad \begin{array}{c} \text{E-PI CONG} \\ \Gamma; \Delta \vDash A_1 \equiv A_2 : \text{type} \\ \Gamma, x : A_1; \Delta \vDash B_1 \equiv B_2 : \text{type} \\ [\Gamma \vDash A_1 : \text{type}] \\ [\Gamma \vDash \Pi^\rho x: A_1. B_1 : \text{type}] \\ [\Gamma \vDash \Pi^\rho x: A_2. B_2 : \text{type}] \\ \hline \Gamma; \Delta \vDash (\Pi^\rho x: A_1. B_1) \equiv (\Pi^\rho x: A_2. B_2) : \text{type} \end{array}$$

$$\begin{array}{c} \text{E-ABS CONG} \\ \Gamma, x : A_1; \Delta \vDash b_1 \equiv b_2 : B \\ [\Gamma \vDash A_1 : \text{type}] \\ (\rho = +) \vee (x \notin \text{fv } b_1) \\ (\rho = +) \vee (x \notin \text{fv } b_2) \\ \hline \Gamma; \Delta \vDash (\lambda^\rho x. b_1) \equiv (\lambda^\rho x. b_2) : \Pi^\rho x: A_1. B \end{array} \quad \begin{array}{c} \text{E-APP CONG} \\ \Gamma; \Delta \vDash a_1 \equiv b_1 : \Pi^+ x: A. B \\ \Gamma; \Delta \vDash a_2 \equiv b_2 : A \\ \hline \Gamma; \Delta \vDash a_1 a_2^+ \equiv b_1 b_2^+ : B\{a_2/x\} \end{array}$$

$$\begin{array}{c} \text{E-IAPP CONG} \\ \Gamma; \Delta \vDash a_1 \equiv b_1 : \Pi^- x: A. B \\ \Gamma \vDash a : A \\ \hline \Gamma; \Delta \vDash a_1 \square^- \equiv b_1 \square^- : B\{a/x\} \end{array} \quad \begin{array}{c} \text{E-PIFST} \\ \Gamma; \Delta \vDash \Pi^\rho x: A_1. B_1 \equiv \Pi^\rho x: A_2. B_2 : \text{type} \\ \hline \Gamma; \Delta \vDash A_1 \equiv A_2 : \text{type} \end{array}$$

$$\begin{array}{c} \text{E-PI SND} \\ \Gamma; \Delta \vDash \Pi^\rho x: A_1. B_1 \equiv \Pi^\rho x: A_2. B_2 : \text{type} \\ \Gamma; \Delta \vDash a_1 \equiv a_2 : A_1 \\ \hline \Gamma; \Delta \vDash B_1\{a_1/x\} \equiv B_2\{a_2/x\} : \text{type} \end{array} \quad \begin{array}{c} \text{E-CPi CONG} \\ \Gamma; \Delta \vDash \phi_1 \equiv \phi_2 \\ \Gamma, c : \phi_1; \Delta \vDash A \equiv B : \text{type} \\ [\Gamma \vDash \phi_1 \text{ ok}] \\ [\Gamma \vDash \forall c: \phi_1. A : \text{type}] \\ [\Gamma \vDash \forall c: \phi_2. B : \text{type}] \\ \hline \Gamma; \Delta \vDash \forall c: \phi_1. A \equiv \forall c: \phi_2. B : \text{type} \end{array}$$

$$\begin{array}{c} \text{E-CABS CONG} \\ \Gamma, c : \phi_1; \Delta \vDash a \equiv b : B \\ [\Gamma \vDash \phi_1 \text{ ok}] \\ \hline \Gamma; \Delta \vDash (\Lambda c. a) \equiv (\Lambda c. b) : \forall c: \phi_1. B \end{array} \quad \begin{array}{c} \text{E-CAPP CONG} \\ \Gamma; \Delta \vDash a_1 \equiv b_1 : \forall c: (a \sim_A b). B \\ \Gamma; \tilde{\Gamma} \vDash a \equiv b : A \\ \hline \Gamma; \Delta \vDash a_1[\bullet] \equiv b_1[\bullet] : B\{\bullet/c\} \end{array}$$

7:28 Eta-Equivalence in Core Dependent Haskell

$$\begin{array}{c}
\text{E-CPISND} \\
\frac{\Gamma; \Delta \vDash \forall c : (a_1 \sim_A a_2). B_1 \equiv \forall c : (a'_1 \sim_{A'} a'_2). B_2 : \text{type} \quad \begin{array}{c} \Gamma; \tilde{\Gamma} \vDash a_1 \equiv a_2 : A \\ \Gamma; \tilde{\Gamma} \vDash a'_1 \equiv a'_2 : A' \end{array}}{\Gamma; \Delta \vDash B_1\{\bullet/c\} \equiv B_2\{\bullet/c\} : \text{type}} \\
\text{E-CAST} \\
\frac{\Gamma; \Delta \vDash a \equiv b : A \quad \Gamma; \Delta \vDash a \sim_A b \equiv a' \sim_{A'} b'}{\Gamma; \Delta \vDash a' \equiv b' : A'} \\
\text{E-EQCONV} \\
\frac{\Gamma; \Delta \vDash a \equiv b : A \quad \Gamma; \tilde{\Gamma} \vDash A \equiv B : \text{type}}{\Gamma; \Delta \vDash a \equiv b : B} \quad \text{E-ISOSND} \\
\frac{\Gamma; \Delta \vDash a \sim_A b \equiv a' \sim_{A'} b'}{\Gamma; \Delta \vDash A \equiv A' : \text{type}} \quad \text{E-ETAREL} \\
\frac{\Gamma \vDash b : \Pi^+ x : A.B \quad a = b x^+}{\Gamma; \Delta \vDash \lambda^+ x. a \equiv b : \Pi^+ x : A.B} \\
\text{E-ETAIRREL} \\
\frac{\Gamma \vDash b : \Pi^- x : A.B \quad a = b \square^-}{\Gamma; \Delta \vDash \lambda^- x. a \equiv b : \Pi^- x : A.B} \quad \text{E-ETAC} \\
\frac{\Gamma \vDash b : \forall c : \phi.B \quad a = b[\bullet]}{\Gamma; \Delta \vDash \Lambda c. a \equiv b : \forall c : \phi.B}
\end{array}$$

$\vDash \Gamma$

(context wellformedness)

$$\begin{array}{c}
\text{E-EMPTY} \\
\frac{}{\vDash \emptyset} \quad \text{E-CONSTM} \\
\frac{\vDash \Gamma \quad \Gamma \vDash A : \text{type} \quad x \notin \text{dom } \Gamma}{\vDash \Gamma, x : A} \quad \text{E-CONSCO} \\
\frac{\Gamma \vDash \phi \text{ ok} \quad c \notin \text{dom } \Gamma}{\vDash \Gamma, c : \phi}
\end{array}$$

$\vDash \Sigma$

(signature wellformedness)

$$\begin{array}{c}
\text{SIG-EMPTY} \\
\frac{}{\vDash \emptyset} \quad \text{SIG-CONSAx} \\
\frac{\vDash \Sigma \quad \emptyset \vDash A : \text{type} \quad \emptyset \vDash a : A \quad F \notin \text{dom } \Sigma}{\vDash \Sigma \cup \{F \sim a : A\}}
\end{array}$$

E Full system specification: System DC type system

$\Gamma \vdash a : A$

(typing)

$$\begin{array}{c}
\text{AN-STAR} \\
\frac{\vdash \Gamma}{\Gamma \vdash \text{type} : \text{type}} \quad \text{AN-VAR} \\
\frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x : A} \quad \text{AN-PI} \\
\frac{\Gamma, x : A \vdash B : \text{type} \quad [\Gamma \vdash A : \text{type}]}{\Gamma \vdash \Pi^\rho x : A.B : \text{type}} \quad \text{AN-ABS} \\
\frac{[\Gamma \vdash A : \text{type}] \quad \Gamma, x : A \vdash a : B \quad (\rho = +) \vee (x \notin \text{fv } |a|)}{\Gamma \vdash \lambda^\rho x : A. a : \Pi^\rho x : A.B} \\
\text{AN-APP} \\
\frac{\Gamma \vdash b : \Pi^\rho x : A.B \quad \Gamma \vdash a : A}{\Gamma \vdash b a^\rho : B\{a/x\}} \quad \text{AN-CONV} \\
\frac{\Gamma \vdash a : A \quad \Gamma; \tilde{\Gamma} \vdash \gamma : A \sim B \quad \Gamma \vdash B : \text{type}}{\Gamma \vdash a \triangleright \gamma : B} \quad \text{AN-CPi} \\
\frac{[\Gamma \vdash \phi \text{ ok}] \quad \Gamma, c : \phi \vdash B : \text{type}}{\Gamma \vdash \forall c : \phi. B : \text{type}} \quad \text{AN-CABS} \\
\frac{\Gamma \vdash \phi \text{ ok} \quad \Gamma, c : \phi \vdash a : B}{\Gamma \vdash \Lambda c : \phi. a : \forall c : \phi. B} \\
\text{AN-CAPP} \\
\frac{\Gamma \vdash a_1 : \forall c : a \sim_{A_1} b. B \quad \Gamma; \tilde{\Gamma} \vdash \gamma : a \sim b}{\Gamma \vdash a_1[\gamma] : B\{\gamma/c\}} \quad \text{AN-FAM} \\
\frac{\vdash \Gamma \quad F \sim a : A \in \Sigma_1 \quad [\emptyset \vdash A : \text{type}]}{\Gamma \vdash F : A}
\end{array}$$

$\Gamma \vdash \phi \text{ ok}$ *(prop wellformedness)*

$$\text{AN-WFF} \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad |A| = |B|}{\Gamma \vdash a \sim_A b \text{ ok}}$$

 $\Gamma; \Delta \vdash \gamma : \phi_1 \sim \phi_2$ *(coercion between props)*

AN-PROPCONG

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : A_1 \sim A_2 \\ \Gamma; \Delta \vdash \gamma_2 : B_1 \sim B_2 \\ \Gamma \vdash A_1 \sim_A B_1 \text{ ok} \\ \Gamma \vdash A_2 \sim_A B_2 \text{ ok} \end{array}}{\Gamma; \Delta \vdash (\gamma_1 \sim_A \gamma_2) : (A_1 \sim_A B_1) \sim (A_2 \sim_A B_2)}$$

AN-CPIFST

$$\frac{\Gamma; \Delta \vdash \gamma : \forall c : \phi_1. A_2 \sim \forall c : \phi_2. B_2}{\Gamma; \Delta \vdash \text{cpiFst } \gamma : \phi_1 \sim \phi_2}$$

AN-ISOSYM

$$\frac{\Gamma; \Delta \vdash \gamma : \phi_1 \sim \phi_2}{\Gamma; \Delta \vdash \text{sym } \gamma : \phi_2 \sim \phi_1}$$

AN-ISOCONV

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma : A \sim B \\ \Gamma \vdash a_1 \sim_A a_2 \text{ ok} \\ \Gamma \vdash a'_1 \sim_B a'_2 \text{ ok} \\ |a_1| = |a'_1| \quad |a_2| = |a'_2| \end{array}}{\Gamma; \Delta \vdash \text{conv } (a_1 \sim_A a_2) \sim_\gamma (a'_1 \sim_B a'_2) : (a_1 \sim_A a_2) \sim (a'_1 \sim_B a'_2)}$$

 $\Gamma; \Delta \vdash \gamma : A \sim B$ *(coercion between types)*

AN-ASSN

$$\frac{\begin{array}{l} \vdash \Gamma \\ c : a \sim_A b \in \Gamma \quad c \in \Delta \end{array}}{\Gamma; \Delta \vdash c : a \sim b}$$

AN-REFL

$$\frac{\Gamma \vdash a : A}{\Gamma; \Delta \vdash \text{refl } a : a \sim a}$$

AN-ERASEEQ

$$\frac{\begin{array}{l} \Gamma \vdash a : A \\ \Gamma \vdash b : B \quad |a| = |b| \\ \Gamma; \tilde{\Gamma} \vdash \gamma : A \sim B \end{array}}{\Gamma; \Delta \vdash (a \mid_{\gamma} b) : a \sim b}$$

AN-SYM

$$\frac{\begin{array}{l} \Gamma \vdash b : B \quad \Gamma \vdash a : A \\ [\Gamma; \tilde{\Gamma} \vdash \gamma_1 : B \sim A] \\ \Gamma; \Delta \vdash \gamma : b \sim a \end{array}}{\Gamma; \Delta \vdash \text{sym } \gamma : a \sim b}$$

AN-TRANS

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : a \sim a_1 \\ \Gamma; \Delta \vdash \gamma_2 : a_1 \sim b \\ [\Gamma \vdash a : A] \\ [\Gamma \vdash a_1 : A_1] \\ [\Gamma; \tilde{\Gamma} \vdash \gamma_3 : A \sim A_1] \end{array}}{\Gamma; \Delta \vdash (\gamma_1; \gamma_2) : a \sim b}$$

AN-BETA

$$\frac{\begin{array}{l} \Gamma \vdash a_1 : B_0 \\ \Gamma \vdash a_2 : B_1 \\ |B_0| = |B_1| \\ \vDash |a_1| > |a_2| \end{array}}{\Gamma; \Delta \vdash \text{red } a_1 a_2 : a_1 \sim a_2}$$

AN-PICONG

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : A_1 \sim A_2 \\ \Gamma, x : A_1; \Delta \vdash \gamma_2 : B_1 \sim B_2 \\ B_3 = B_2\{x \triangleright \text{sym } \gamma_1/x\} \\ \Gamma \vdash \Pi^\rho x : A_1. B_1 : \text{type} \\ \Gamma \vdash \Pi^\rho x : A_2. B_3 : \text{type} \\ \Gamma \vdash (\Pi^\rho x : A_1. B_2) : \text{type} \end{array}}{\Gamma; \Delta \vdash \Pi^\rho x : \gamma_1. \gamma_2 : (\Pi^\rho x : A_1. B_1) \sim (\Pi^\rho x : A_2. B_3)}$$

7:30 Eta-Equivalence in Core Dependent Haskell

AN-ABSCONG

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : A_1 \sim A_2 \\ \Gamma, x : A_1; \Delta \vdash \gamma_2 : b_1 \sim b_2 \\ b_3 = b_2\{x \triangleright \mathbf{sym} \gamma_1/x\} \\ [\Gamma \vdash A_1 : \mathbf{type}] \\ \Gamma \vdash A_2 : \mathbf{type} \\ (\rho = +) \vee (x \notin \mathbf{fv} |b_1|) \\ (\rho = +) \vee (x \notin \mathbf{fv} |b_3|) \\ [\Gamma \vdash (\lambda^\rho x : A_1.b_2) : B] \end{array}}{\Gamma; \Delta \vdash (\lambda^\rho x : \gamma_1.\gamma_2) : (\lambda^\rho x : A_1.b_1) \sim (\lambda^\rho x : A_2.b_3)}$$

AN-APPCONG

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : a_1 \sim b_1 \\ \Gamma; \Delta \vdash \gamma_2 : a_2 \sim b_2 \\ \Gamma \vdash a_1 a_2^\rho : A \\ \Gamma \vdash b_1 b_2^\rho : B \\ [\Gamma; \tilde{\Gamma} \vdash \gamma_3 : A \sim B] \end{array}}{\Gamma; \Delta \vdash \gamma_1 \gamma_2^\rho : a_1 a_2^\rho \sim b_1 b_2^\rho}$$

AN-PIsND

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : \Pi^\rho x : A_1.B_1 \sim \Pi^\rho x : A_2.B_2 \\ \Gamma; \Delta \vdash \gamma_2 : a_1 \sim a_2 \\ \Gamma \vdash a_1 : A_1 \\ \Gamma \vdash a_2 : A_2 \end{array}}{\Gamma; \Delta \vdash \gamma_1 @ \gamma_2 : B_1\{a_1/x\} \sim B_2\{a_2/x\}}$$

AN-PIFST

$$\frac{\Gamma; \Delta \vdash \gamma : \Pi^\rho x : A_1.B_1 \sim \Pi^\rho x : A_2.B_2}{\Gamma; \Delta \vdash \mathbf{piFst} \gamma : A_1 \sim A_2}$$

AN-CPICONG

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : \phi_1 \sim \phi_2 \\ \Gamma, c : \phi_1; \Delta \vdash \gamma_3 : B_1 \sim B_2 \\ B_3 = B_2\{c \triangleright \mathbf{sym} \gamma_1/c\} \\ \Gamma \vdash \forall c : \phi_1.B_1 : \mathbf{type} \\ [\Gamma \vdash \forall c : \phi_2.B_3 : \mathbf{type}] \\ \Gamma \vdash \forall c : \phi_1.B_2 : \mathbf{type} \end{array}}{\Gamma; \Delta \vdash (\forall c : \gamma_1.\gamma_3) : (\forall c : \phi_1.B_1) \sim (\forall c : \phi_2.B_3)}$$

AN-CABSCONG

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : \phi_1 \sim \phi_2 \\ \Gamma, c : \phi_1; \Delta \vdash \gamma_3 : a_1 \sim a_2 \\ a_3 = a_2\{c \triangleright \mathbf{sym} \gamma_1/c\} \\ \Gamma \vdash (\Lambda c : \phi_1.a_1) : \forall c : \phi_1.B_1 \\ \Gamma \vdash (\Lambda c : \phi_2.a_3) : \forall c : \phi_2.B_2 \\ \Gamma \vdash (\Lambda c : \phi_1.a_2) : B \\ \Gamma; \tilde{\Gamma} \vdash \gamma_4 : \forall c : \phi_1.B_1 \sim \forall c : \phi_2.B_2 \end{array}}{\Gamma; \Delta \vdash (\lambda c : \gamma_1.\gamma_3 @ \gamma_4) : (\Lambda c : \phi_1.a_1) \sim (\Lambda c : \phi_2.a_3)}$$

AN-CAPPCONG

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : a_1 \sim b_1 \\ \Gamma; \tilde{\Gamma} \vdash \gamma_2 : a_2 \sim b_2 \\ \Gamma; \tilde{\Gamma} \vdash \gamma_3 : a_3 \sim b_3 \\ \Gamma \vdash a_1[\gamma_2] : A \\ \Gamma \vdash b_1[\gamma_3] : B \\ [\Gamma; \tilde{\Gamma} \vdash \gamma_4 : A \sim B] \end{array}}{\Gamma; \Delta \vdash \gamma_1(\gamma_2, \gamma_3) : a_1[\gamma_2] \sim b_1[\gamma_3]}$$

AN-CPIsND

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : (\forall c_1 : a \sim_A a'.B_1) \sim (\forall c_2 : b \sim_B b'.B_2) \\ \Gamma; \tilde{\Gamma} \vdash \gamma_2 : a \sim a' \\ \Gamma; \tilde{\Gamma} \vdash \gamma_3 : b \sim b' \end{array}}{\Gamma; \Delta \vdash \gamma_1 @ (\gamma_2 \sim \gamma_3) : B_1\{\gamma_2/c_1\} \sim B_2\{\gamma_3/c_2\}}$$

AN-CAST

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash \gamma_1 : a \sim a' \\ \Gamma; \Delta \vdash \gamma_2 : a \sim_A a' \sim b \sim_B b' \end{array}}{\Gamma; \Delta \vdash \gamma_1 \triangleright \gamma_2 : b \sim b'}$$

AN-ISOSND

$$\frac{\Gamma; \Delta \vdash \gamma : (a \sim_A a') \sim (b \sim_B b')}{\Gamma; \Delta \vdash \mathbf{isoSnd} \gamma : A \sim B}$$

AN-ETA

$$\frac{\begin{array}{l} \Gamma \vdash b : \Pi^\rho x : A.B \\ a = b x^\rho \end{array}}{\Gamma; \Delta \vdash \mathbf{eta} b : (\lambda^\rho x : A.a) \sim b}$$

$$\frac{\text{AN-ETAC} \quad \Gamma \vdash b : \forall c : \phi. B \quad a = b[c]}{\Gamma; \Delta \vdash \mathbf{eta} \, b : (\Lambda c : \phi. a) \sim b}$$

 $\boxed{\vdash \Gamma}$ *(context wellformedness)*

$$\frac{\text{AN-EMPTY}}{\vdash \emptyset} \quad \frac{\text{AN-CONSTM} \quad \vdash \Gamma \quad \Gamma \vdash A : \text{type} \quad x \notin \text{dom } \Gamma}{\vdash \Gamma, x : A} \quad \frac{\text{AN-CONSCo} \quad \vdash \Gamma \quad \Gamma \vdash \phi \text{ ok} \quad c \notin \text{dom } \Gamma}{\vdash \Gamma, c : \phi}$$

 $\boxed{\vdash \Sigma}$ *(signature wellformedness)*

$$\frac{\text{AN-SIG-EMPTY}}{\vdash \emptyset} \quad \frac{\text{AN-SIG-CONSAx} \quad \vdash \Sigma \quad \emptyset \vdash A : \text{type} \quad \emptyset \vdash a : A \quad F \notin \text{dom } \Sigma}{\vdash \Sigma \cup \{F \sim a : A\}}$$

Coherence for Monoidal Groupoids in HoTT

Stefano Piceghello 

Department of Informatics and Department of Mathematics, University of Bergen, Norway
stefano.piceghello@uib.no

Abstract

We present a proof of coherence for monoidal groupoids in homotopy type theory. An important role in the formulation and in the proof of coherence is played by groupoids with a free monoidal structure; these can be represented by 1-truncated higher inductive types, with constructors freely generating their defining objects, natural isomorphisms and commutative diagrams. All results included in this paper have been formalised in the proof assistant Coq.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Constructive mathematics

Keywords and phrases homotopy type theory, coherence, monoidal categories, groupoids, higher inductive types, formalisation, Coq

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.8

Supplementary Material Formalised proofs are available at <https://github.com/spiceghello/FSMG>.

Acknowledgements The author wishes to thank Kristian S. Alfsvåg, Marc Bezem, Floris van Doorn, Bjørn Ian Dundas, Peter Dybjer, Favonia and Håkon R. Gylterud for providing valuable insight on the topic and the anonymous reviewers for constructive comments, and acknowledges the support of the Centre for Advanced Study (CAS) in Oslo, Norway, which funded and hosted the research project *Homotopy Type Theory and Univalent Foundations* during the 2018/19 academic year.

1 Introduction

Homotopy type theory (HoTT) [21] is a version of Martin-Löf type theory, enhanced with a definition of identity types that allows the interpretation of terms in a type as points in spaces, and terms in identity types as paths; iterating the construction of identity types promotes the interpretation of types as ∞ -groupoids [17, 21, 3]. One of the key features of HoTT is the use of higher inductive types (HITs) [18, 21, 8], which integrate inductive constructions with the higher groupoid structure of types. This has led to the formalisation of numerous results and computations in homotopy theory (see e.g. [21, Chapter 8] for a nonexhaustive list), which have largely been checked using proof assistants such as Coq [12, 11], Agda [7] and Lean [9]. In this paper we employ the functionalities of HITs to formalise in HoTT the result in category theory known as coherence for monoidal categories.

A (weak) *monoidal category* consists of a category \mathcal{C} together with a *monoidal structure*, i.e. a bifunctor $\bullet : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ that serves as product, an object $e \in \text{ob}(\mathcal{C})$ that serves as unit for the product, and natural isomorphisms describing associativity and unitality of the product (w.r.t. the unit object) and making two classes of diagrams – namely, the *coherence diagrams* in Figures 1a and 1b – commute. A monoidal category is said to be *strict* if the associativity and unitality natural isomorphisms are identities. Monoidal categories satisfy a theorem of coherence, which states that every monoidal category is monoidally equivalent to a strict one; an equivalent formulation [14, 19] is the following:

► **Statement 1.** *Every diagram in a free monoidal category commutes.*



© Stefano Piceghello;

licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 8; pp. 8:1–8:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This means that all (nontrivial) diagrams in a free monoidal category can be expressed as combinations of instances of the coherence diagrams and of the naturality diagrams of associativity and unitality.

Different proofs of Statement 1 have been formalised in intensional MLTT in [4, 5] using the proof assistant ALF [20] (with Axiom K) and later in [1] using HOL [10]; there, a category is given by a set of objects and a family of Hom-sets. After noticing that a free monoidal category is a groupoid (since all its arrows are built upon instances of the natural isomorphisms defining the monoidal structure, and hence they are invertible), a version of the same statement can be proved in HoTT by exploiting the mentioned groupoid structure of types (Theorem 7). Indeed, monoidal groupoids can be represented by 1-types, using the correspondence between objects in a groupoid and terms in the type, arrows and paths,¹ and commutative diagrams and 2-paths, where monoidality refers to the presence in the type of the necessary terms, paths and 2-paths to define a monoidal structure. Moreover, HITs offer a way to describe objects satisfying certain universal properties, allowing us to define types that can be interpreted as *free* monoidal groupoids.

Since every path in a type has an inverse, this framework only allows us to express groupoids and not categories; however, a formulation of Statement 1 for groupoids is, from a mathematical standpoint, equivalent to the original one, as the free objects in the categories of monoidal groupoids and of monoidal categories coincide. The choice of proving coherence as formulated in Statement 1, without referring to strict monoidal structures, is also due to the design of the theory, as discussed in Section 5.

The results included in this paper have been formalised using the Coq proof assistant (see Section 6 for further details); the code is available as supplementary material.

Background and Notation

We assume familiarity with the basics of HoTT; our main reference is [21], from which we largely borrow our notation. In particular:

- we will use the symbol \equiv for judgmental equality and $:\equiv$ for definitions;
- we will not distinguish, in the notation, between different members of an assumed hierarchy of universes, and will instead denote them uniformly by \mathcal{U} ;
- for a family of types $B : A \rightarrow \mathcal{U}$, dependent functions are denoted by $f :\equiv (x \mapsto f(x)) : \Pi(x : A).B(x)$; the identity function is indicated by $\text{id}_A :\equiv \text{id} : A \rightarrow A$; dependent pairs are denoted by $\langle a, b \rangle : \Sigma(x : A).B(x)$; terms in nested Σ -types are denoted by tuples $\langle a, b, c, \dots \rangle$;
- identity types are denoted by $x =_A y$ or simply $x = y$ for $x, y : A$; their terms are called *paths* in A and their elimination principle is called *path induction*; identity (reflexivity) paths, which inductively generate identity types, are denoted by $\text{refl}_x :\equiv \text{refl} : x = x$; terms $r : p = q$, where p and q are paths in a type, are called *2-paths*, and so on;
- the inverse of a path p is denoted by p^{-1} ; the concatenation of paths $p : x = y$ and $q : y = z$ is denoted by $p \cdot q : x = z$; this operation is provably associative and unital with respect to the identity path and it satisfies inverse laws, giving rise to the umbrella-expression “path algebra” to encompass all proofs of existence of 2-paths of the sort;
- the action of $f : A \rightarrow B$ on a path $p : x =_A y$ is denoted by $\text{ap}_f(p) : f(x) =_B f(y)$; functoriality of ap will also be referred to as path algebra;

¹ Observe that, by their very nature, the categories we are describing are univalent.

- given a family of types $P : A \rightarrow \mathcal{U}$ and a path $p : x =_A y$, the transport of terms in $P(x)$ along p is denoted by $p_*^P : \equiv p_* : P(x) \rightarrow P(y)$; the action of $f : \Pi(a : A).P(a)$ on p is indicated by $\mathbf{ap}_f(p) : p_*(f(x)) =_{P(y)} f(y)$; a term in the identity type $p_*(u) = v$ is called a *pathover* [15];
- several results are assumed about transporting in families of paths; in particular, we will implicitly use that, given functions $f, g : A \rightarrow B$ and paths $p : x =_A y$ and $q : f(x) =_B g(x)$, there is a (2-)pathover of type $p_*^{(a \mapsto f(a)=g(a))}(q) = \mathbf{ap}_f(p)^{-1} \cdot q \cdot \mathbf{ap}_g(p)$;
- pointwise equalities of functions f and $g : A \rightarrow B$ are called *homotopies* and denoted by $f \sim g : \equiv \Pi(x : A).f(x) = g(x)$; if $h : B \rightarrow A$ and $f \circ h \sim \text{id}_B$, f is said to be a *retraction* of h and B a *retract* of A ;
- we denote by $A \simeq B : \equiv \Sigma(f : A \rightarrow B, g : B \rightarrow A).(g \circ f \sim \text{id}_A) \times (f \circ g \sim \text{id}_B)$ the type of *equivalences* between A and B ; f and g are said to be half adjoint in such an equivalence;
- the prefix “0-” or “1-” for types refers to their *truncation level*; a 0-type is a type A such that there is a term in $\Pi(x, y : A).\Pi(p, q : x = y).p = q$, embodying the notion of a *set* of terms, while A is a 1-type (i.e. a *groupoid*) if all its identity types are 0-types; every 0-type is a 1-type; the property of A being a 1-type is denoted by $\text{IsTrunc}_1(A)$;
- as mentioned, the theory assumes HITs; in the presentation we will informally call “computation rules” also the assumed identities involving the (dependent) application of the elimination principle of a HIT on higher constructors, although no computation takes place [21, Chapter 6].

We will, moreover, refer to a (2-)path as “trivial” if it is either the identity path or it can be obtained by path algebra.

In figures presenting 2-paths, we choose to display paths $p : x = y$ as arrows $x \rightarrow y$, both to preserve the analogy with categories and to keep track of the endpoints; in such diagrams, all arrows are invertible, as all paths are. A dotted line denotes instead judgmental equality. Figures relevant to proofs are included in Appendix A.

2 Monoidal Groupoids

In this section we will provide the definitions of the objects of our study, building a categorical framework within which to formulate the theorem of coherence for monoidal groupoids.

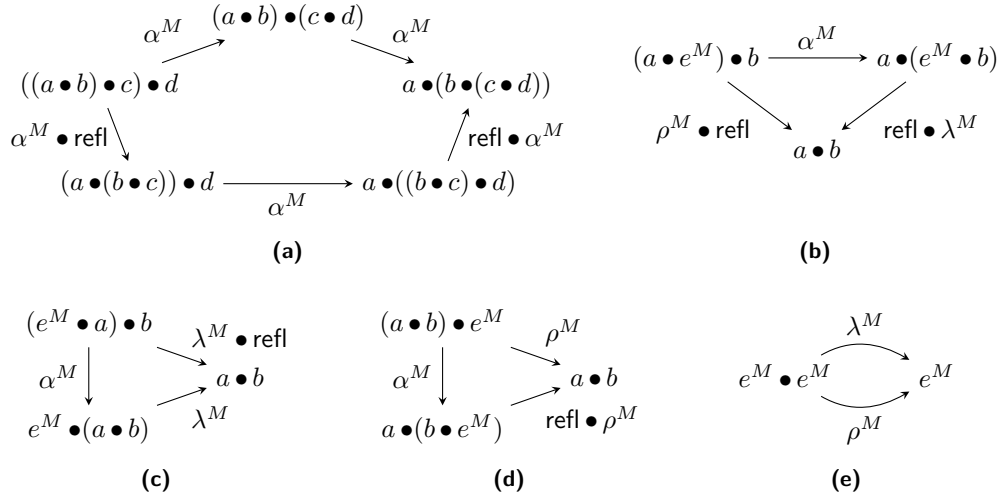
► **Definition 2.** A *groupoid* is the data given by a type G and a proof that G is a 1-type; we call $\text{Gpd} : \equiv \Sigma(G : \mathcal{U}).\text{IsTrunc}_1(G)$ the subuniverse of 1-types in the universe \mathcal{U} . A (groupoid) *functor* F between groupoids is simply a function between the underlying (1-)types; a *natural isomorphism* between two functors is a homotopy between the functions.

In Definition 2 we bestow the title of “functor” on simple functions, as the functorial action on paths is provided by \mathbf{ap} . We will use the same notation for a groupoid $G : \text{Gpd}$ and its underlying type $G : \mathcal{U}$. This framework allows us to represent categories with all arrows invertible (i.e. *groupoids*) as 1-types, with paths taking the role of arrows and 2-paths that of commutative diagrams.²

► **Definition 3.** For a type M , the type $\text{MonStr}(M)$ of *monoidal structures* on M is the Σ -type encoding the following data:

² This translation implies a “relaxation” of certain strict categorical properties: for example, associativity of the composition of the arrows in a category is strict, while associativity of concatenation of paths only holds up to a coherent choice of higher paths. Moreover, we remark that, given a groupoid $G : \text{Gpd}$, we do not have access to the discrete subcategory of its objects.

8:4 Coherence for Monoidal Groupoids in HoTT



■ **Figure 1** (a) and (b): coherence diagrams \triangleleft^M and \triangleleft^M , respectively, where α^M , λ^M and ρ^M are associativity and left and right unitality morphisms; (c), (d) and (e): coherence diagrams derivable from \triangleleft^M , \triangleleft^M and naturality of α^M , λ^M and ρ^M ; the derivation is shown in Figures 5 and 6. Here \bullet^M is denoted by \bullet for clarity.

- $e^M : M$ (unit);
 - $\bullet^M : M \rightarrow M \rightarrow M$ (monoidal product, infix notation);
 - $\alpha^M : \Pi(a, b, c : M). (a \bullet^M b) \bullet^M c = a \bullet^M (b \bullet^M c)$ (associativity);
 - $\lambda^M : \Pi(b : M). e^M \bullet^M b = b$ (left unitality);
 - $\rho^M : \Pi(a : M). a \bullet^M e^M = a$ (right unitality);
 - families \triangleleft^M and \triangleleft^M of 2-paths filling the coherence diagrams in Figures 1a and 1b.
- The type of **monoidal groupoids** is defined as $\text{MonGpd} := \Sigma(M : \text{Gpd}). \text{MonStr}(M)$.

We will use the same notation for a monoidal groupoid M and its carrier. The functorial action of \bullet^M on paths and 2-paths, denoted in this paper by the same symbol, is given by the following functions, each defined by path induction (on both arguments):

$$\begin{aligned} \bullet^M &: (a_1 =_M b_1) \rightarrow (a_2 =_M b_2) \rightarrow (a_1 \bullet^M a_2 =_M b_1 \bullet^M b_2), \\ \bullet^M &: (p =_{(a_1=b_1)} p') \rightarrow (q =_{(a_2=b_2)} q') \rightarrow (p \bullet^M q =_{(a_1 \bullet^M a_2 =_M b_1 \bullet^M b_2)} p' \bullet^M q'). \end{aligned}$$

We emphasise that the given definition of a monoidal structure only pertains the levels of coherence for associativity and unitality that might be present in a (non-higher) groupoid (“1-coherent” monoidal structure, [6]), i.e., no higher diagrams need to be described, as they are present already.

By path induction, α^M , λ^M and ρ^M are natural in all their arguments. Moreover, the 2-paths in Figures 1c to 1e can be derived by instances of the coherence diagrams \triangleleft^M and \triangleleft^M , together with naturality of associativity and unitality [13], as shown in Figures 5 and 6.

► **Definition 4.** The type $\text{MonGpd}(M, N)$ of (strong) **monoidal functors** between two monoidal groupoids $\langle M, e^M, \bullet^M, \dots \rangle$ and $\langle N, e^N, \bullet^N, \dots \rangle$ is defined as the Σ -type encoding the following data:

- a functor $F : M \rightarrow N$;
- a path $F_0 : e^N = F(e^M)$ and a family of paths $F_2 : \Pi(a, b : M). F(a) \bullet^N F(b) = F(a \bullet^M b)$;
- families of 2-paths F_α , F_λ and F_ρ , as depicted in Figure 2.

$$\begin{array}{ccc}
(F(a) \bullet^N F(b)) \bullet^N F(c) & \xrightarrow{\alpha^N} & F(a) \bullet^N (F(b) \bullet^N F(c)) \\
F_2 \bullet^N \text{refl} \downarrow & & \downarrow \text{refl} \bullet^N F_2 \\
F(a \bullet^M b) \bullet^N F(c) & & F(a) \bullet^N F(b \bullet^M c) \\
F_2 \downarrow & & \downarrow F_2 \\
F((a \bullet^M b) \bullet^M c) & \xrightarrow{\text{ap}_F(\alpha^M)} & F(a \bullet^M (b \bullet^M c))
\end{array}$$

$$\begin{array}{ccc}
e^N \bullet^N F(b) & \xrightarrow{\lambda^N} & F(b) & & F(a) \bullet^N e^N & \xrightarrow{\rho^N} & F(a) \\
F_0 \bullet^N \text{refl} \downarrow & & \uparrow \text{ap}_F(\lambda^M) & & \text{refl} \bullet^N F_0 \downarrow & & \uparrow \text{ap}_F(\rho^M) \\
F(e^M) \bullet^N F(b) & \xrightarrow{F_2} & F(e^M \bullet^M b) & & F(a) \bullet^N F(e^M) & \xrightarrow{F_2} & F(a \bullet^M e^M)
\end{array}$$

■ **Figure 2** Coherence conditions $F_\alpha(a, b, c)$, $F_\lambda(b)$ and $F_\rho(a)$ (respectively: top, bottom left and bottom right) for monoidal functors, for $a, b, c : M$.

$$\begin{array}{ccc}
& & e^N & & \\
& F_0 \swarrow & & \searrow G_0 & \\
F(e^M) & \xrightarrow{\theta(e^M)} & G(e^M) & & \\
& & & &
\end{array}$$

$$\begin{array}{ccc}
F(a) \bullet^N F(b) & \xrightarrow{F_2} & F(a \bullet^M b) \\
\theta(a) \bullet^N \theta(b) \downarrow & & \downarrow \theta(a \bullet^M b) \\
G(a) \bullet^N G(b) & \xrightarrow{G_2} & G(a \bullet^M b)
\end{array}$$

■ **Figure 3** Coherence conditions θ_0 and $\theta_2(a, b)$ for monoidal natural isomorphisms, for $a, b : M$.

We will use the same notation for a monoidal functor $F : \text{MonGpd}(M, N)$ and its underlying function. This implementation allows us to provide sound definitions of identity and composition of monoidal functors:

$$\begin{aligned}
(\text{id}_M)_0 &::= \text{refl} : e_M = e_M \\
(\text{id}_M)_2(a, b) &::= \text{refl} : a \bullet^M b = a \bullet^M b, \text{ and} \\
(G \circ F)_0 &::= G_0 \cdot \text{ap}_G(F_0) : e_P = G(F(e_M)) \\
(G \circ F)_2(a, b) &::= G_2(F(a), F(b)) \cdot \text{ap}_G(F_2(a, b)) : G(F(a)) \bullet^P G(F(b)) = G(F(a \bullet^M b)),
\end{aligned}$$

for $F : \text{MonGpd}(M, N)$, $G : \text{MonGpd}(N, P)$ and $a, b : M$; we omit here the 2-paths $(G \circ F)_\alpha$, $(G \circ F)_\lambda$ and $(G \circ F)_\rho$ (appearing in the formalisation), while the corresponding ones for identity monoidal functors are trivial.

► **Definition 5.** *The type $\text{MonFun}_{M,N}(F, G)$ of **monoidal natural isomorphisms** between monoidal functors $F, G : \text{MonGpd}(M, N)$ is the Σ -type encoding a natural isomorphism $\theta : F \sim G$ between the underlying functors, together with a 2-path θ_0 and a family of 2-paths θ_2 , as shown in Figure 3.*

We will use the same notation for a monoidal natural isomorphism $\theta : \text{MonFun}_{M,N}(F, G)$ and its underlying homotopy. If $F : \text{MonGpd}(M, N)$ and $G : \text{MonGpd}(N, M)$, the underlying homotopies in $\eta : \text{MonFun}_{M,M}(\text{id}_M, G \circ F)$ and $\varepsilon : \text{MonFun}_{N,N}(F \circ G, \text{id}_N)$ prove that the functions underlying F and G are half adjoint in an equivalence between (the carriers of) M and N ; this will be called a **monoidal equivalence** and denoted by $M \simeq N$.

8:6 Coherence for Monoidal Groupoids in HoTT

$$\begin{array}{ccc}
 \text{MonGpd}(F(X), M) & \xrightarrow{\phi_{X,M}} & (X \rightarrow M) \\
 H \circ - \downarrow & & \downarrow H \circ - \\
 \text{MonGpd}(F(X), N) & \xrightarrow{\phi_{X,N}} & (X \rightarrow N)
 \end{array}
 \qquad
 \begin{array}{ccc}
 (X \rightarrow M) & \xrightarrow{\psi_{X,M}} & \text{MonGpd}(F(X), M) \\
 - \circ h \downarrow & & \downarrow - \circ F(h) \\
 (Y \rightarrow M) & \xrightarrow{\psi_{Y,M}} & \text{MonGpd}(F(Y), M)
 \end{array}$$

(a) Naturality of ϕ in M : the diagram commutes for every $H : \text{MonGpd}(M, N)$, i.e. there is a homotopy $H \circ \phi_{X,M}(G) \sim \phi_{X,N}(H \circ G)$ for every $G : \text{MonGpd}(F(X), M)$.

(b) Naturality of ψ in X : the diagram commutes for every $h : Y \rightarrow X$, i.e. there is a term in $\text{MonFun}_{F(Y), M}(\psi_{X,M}(g) \circ F(h), \psi_{Y,M}(g \circ h))$ for every $g : X \rightarrow M$.

■ **Figure 4** Naturality conditions for ϕ and ψ in Definition 6.

► **Definition 6.** A **functor** from the universe \mathcal{U} of types to monoidal groupoids consists of a function term $F : \mathcal{U} \rightarrow \text{MonGpd}$, together with a function between function types

$$\vec{F} : \Pi(X, Y : \mathcal{U}).(X \rightarrow Y) \rightarrow \text{MonGpd}(F(X), F(Y))$$

respecting identity and composition, i.e. terms

$$\begin{aligned}
 F_{\text{id}} &: \Pi(X : \mathcal{U}).\text{MonFun}_{F(X), F(X)}(\vec{F}(\text{id}_X), \text{id}_{F(X)}) \text{ and} \\
 F_{\circ} &: \Pi(X, Y, Z : \mathcal{U}, f : X \rightarrow Y, g : Y \rightarrow Z).\text{MonFun}_{F(X), F(Z)}(\vec{F}(g) \circ \vec{F}(f), \vec{F}(g \circ f)).
 \end{aligned}$$

We will refer to a function $F : \mathcal{U} \rightarrow \text{MonGpd}$ as a “functor” if the remaining data is implied. Such a functor is **free** if there are:

- a function $\phi : \Pi(X : \mathcal{U}).\Pi(M : \text{MonGpd}).\text{MonGpd}(F(X), M) \rightarrow X \rightarrow M$, natural in M , i.e. the diagram in Figure 4a commutes for every $H : \text{MonGpd}(M, N)$;
- a function $\psi : \Pi(X : \mathcal{U}).\Pi(M : \text{MonGpd}).(X \rightarrow M) \rightarrow \text{MonGpd}(F(X), M)$, natural in X , i.e. the diagram in Figure 4b commutes for every $h : Y \rightarrow X$;
- a family of homotopies $\theta : \Pi(X : \mathcal{U}).\Pi(M : \text{MonGpd}).\phi_{X,M} \circ \psi_{X,M} \sim \text{id}_{X \rightarrow M}$;
- a family of monoidal natural isomorphisms

$$\begin{aligned}
 \chi &: \Pi(X : \mathcal{U}).\Pi(M : \text{MonGpd}).\Pi(G : \text{MonGpd}(F(X), M)). \\
 &\quad \text{MonFun}_{F(X), M}(\psi_{X,M}(\phi_{X,M}(G)), G).
 \end{aligned}$$

If $X : \mathcal{U}$, the monoidal groupoid $F(X)$ is said to be **freely generated** by X .

One can recognise, in the data listed above for the definition of a free functor, the requirements needed to verify that F is left adjoint to the forgetful functor to types, which in this case is the composition of the projections $\text{MonGpd} \rightarrow \text{Gpd}$ and $\text{Gpd} \rightarrow \mathcal{U}$.

Focussing in Definition 6 on free monoidal groupoids generated by 0-types, Statement 1 can be then formulated as follows:

► **Theorem 7** (Coherence for monoidal groupoids). A free functor $\text{FMG} : \mathcal{U} \rightarrow \text{MonGpd}$ exists and, for every 0-type X , the carrier of $\text{FMG}(X)$ is a 0-type.

Indeed, Theorem 7 both ensures that the construction of a free monoidal groupoid (over a set) is possible, and shows that every diagram in such a groupoid commutes, i.e. that there is a 2-path between every two paths sharing endpoints.

Coherence will be achieved by means of two functors: one will be easily proved to be free (Section 3); the other one (list) to produce monoidal groupoids that are also 0-types. We will show that these two functors produce equivalent types (Section 4).

3 Free Monoidal Groupoids

Higher inductive types allow us to define a functor $\text{FMG} : \mathcal{U} \rightarrow \text{MonGpd}$ that contains the proof of its freeness in the elimination principle of the types it produces.

► **Definition 8** (FMG). *Given a type $X : \mathcal{U}$, the HIT $\text{FMG}(X)$ is defined with the following constructors:*

$$\begin{aligned} \text{FMG}(X) &::= e : \text{FMG}(X) \mid \iota : X \rightarrow \text{FMG}(X) \mid \bullet : \text{FMG}(X) \rightarrow \text{FMG}(X) \rightarrow \text{FMG}(X) \\ &\quad \mid \alpha : \Pi(a, b, c : \text{FMG}(X)).(a \bullet b) \bullet c = a \bullet (b \bullet c) \\ &\quad \mid \lambda : \Pi(b : \text{FMG}(X)).e \bullet b = b \mid \rho : \Pi(a : \text{FMG}(X)).a \bullet e = a \\ &\quad \mid \diamond : \dots \mid \nabla : \dots \mid T : \text{IsTrunc}_1(\text{FMG}(X)), \end{aligned}$$

where \diamond and ∇ are families of 2-path constructors corresponding to the coherence diagrams in Figures 1a and 1b.

For any type X , $\text{FMG}(X)$ is a monoidal groupoid, with the monoidal structure provided by the constructors of the HIT.

► **Remark 9.** The elimination rule of $\text{FMG}(X)$ states that, given a family $P : \text{FMG}(X) \rightarrow \mathcal{U}$ together with terms:

- $e' : P(e)$; $\iota' : \Pi(x : X).P(\iota(x))$; $\bullet' : \Pi(a, b : \text{FMG}(X)).P(a) \rightarrow P(b) \rightarrow P(a \bullet b)$ (infix; we will keep the arguments a and b implicit in the notation);
- a family α' witnessing, for every $a, b, c : \text{FMG}(X)$ and $a' : P(a)$, $b' : P(b)$, $c' : P(c)$, a pathover $(\alpha_{a,b,c})_*^P ((a' \bullet' b') \bullet' c') = a' \bullet' (b' \bullet' c')$; similarly defined families of pathovers λ' and ρ' ;
- appropriate families of 2-pathovers \diamond' and ∇' ;
- $T' : \Pi(w : \text{FMG}(X)).\text{IsTrunc}_1(P(w))$,

there is a function $\text{ind} ::= \text{ind}_{\text{FMG}}(e', \iota', \bullet', \dots) : \Pi(w : \text{FMG}(X)).P(w)$, such that

$$\begin{aligned} \text{ind}(e) &\equiv e', & \text{apd}_{\text{ind}}(\alpha_{a,b,c}) &= \alpha'_{\text{ind}(a), \text{ind}(b), \text{ind}(c)}, \\ \text{ind}(\iota(x)) &\equiv \iota'(x), & \text{apd}_{\text{ind}}(\lambda_b) &= \lambda'_{\text{ind}(b)}, \\ \text{ind}(a \bullet b) &\equiv \text{ind}(a) \bullet' \text{ind}(b), & \text{apd}_{\text{ind}}(\rho_a) &= \rho'_{\text{ind}(a)}, \end{aligned}$$

for all $x : X$ and $a, b, c : \text{FMG}(X)$. When instantiated to constant families, it provides the following recursors: given a monoidal groupoid $\langle M, e', \bullet', \alpha', \dots \rangle$ and a function $\iota' : X \rightarrow M$, there is a function $\text{rec} ::= \text{rec}_{\text{FMG}}(e', \iota', \bullet', \dots) : \text{FMG}(X) \rightarrow M$ such that

$$\begin{aligned} \text{rec}(e) &\equiv e', & \text{ap}_{\text{rec}}(\alpha_{a,b,c}) &= \alpha'_{\text{rec}(a), \text{rec}(b), \text{rec}(c)}, \\ \text{rec}(\iota(x)) &\equiv \iota'(x), & \text{ap}_{\text{rec}}(\lambda_b) &= \lambda'_{\text{rec}(b)}, \\ \text{rec}(a \bullet b) &\equiv \text{rec}(a) \bullet' \text{rec}(b), & \text{ap}_{\text{rec}}(\rho_a) &= \rho'_{\text{rec}(a)}, \end{aligned} \tag{3.1}$$

for all $x : X$ and $a, b, c : \text{FMG}(X)$; this is also the underlying function of a monoidal functor, with rec_0 and $\text{rec}_2(a, b)$ being identity paths for every $a, b : \text{FMG}(X)$ and rec_α , rec_λ , rec_ρ given by the computation rules in (3.1).

The construction FMG is functorial, as shown in the following lemma.

► **Lemma 10.** *The function $\text{FMG} : \mathcal{U} \rightarrow \text{MonGpd}$, $X \mapsto \langle \langle \text{FMG}(X), T \rangle, e, \bullet, \alpha, \lambda, \rho, \diamond, \nabla \rangle$ is a functor.*

8:8 Coherence for Monoidal Groupoids in HoTT

Proof. Let $f : X \rightarrow Y$ be a function of types. By Remark 9, a function $\iota' : X \rightarrow \text{FMG}(Y)$ is sufficient to define a monoidal functor $\text{FMG}(f) : \text{MonGpd}(\text{FMG}(X), \text{FMG}(Y))$; this can be given by $\iota' := \iota \circ f$. The proof that FMG respects identity and composition is given in detail in the Coq formalisation included as supplementary material for this paper, and it is also provided by the elimination rule of $\text{FMG}(X)$. By way of example, given a type $X : \mathcal{U}$, a monoidal natural isomorphism

$$\text{FMG}_{\text{id}} : \text{MonFun}_{\text{FMG}(X), \text{FMG}(X)}(\text{FMG}(\text{id}_X), \text{id}_{\text{FMG}(X)})$$

has as underlying homotopy

$$\text{FMG}_{\text{id}} : \Pi(w : \text{FMG}(X)). \text{rec}_{\text{FMG}}(\text{FMG}(X), e, \iota, \bullet, \dots)(w) = w$$

the function $\text{FMG}_{\text{id}} := \text{ind}_{\text{FMG}}(e', \iota', \bullet', \dots)$, with:

- $e' := \text{refl} : e = e$;
- $\iota'(x) := \text{refl} : \iota(x) = \iota(x)$ for every $x : X$;
- $a' \bullet' b' : \text{rec}_{\text{FMG}}(a) \bullet \text{rec}_{\text{FMG}}(b) = a \bullet b$, for $a, b : \text{FMG}(X)$, $a' : \text{rec}_{\text{FMG}}(a) = a$ and $b' : \text{rec}_{\text{FMG}}(b) = b$, is given recursively by $a' \bullet' b'$ (so that $\text{FMG}_{\text{id}}(a \bullet b)$ will compute to $\text{FMG}_{\text{id}}(a) \bullet \text{FMG}_{\text{id}}(b)$);
- the other required terms are obtained by naturality of associativity and unitality, together with the computation rules of rec_{FMG} .

With this definition, the diagrams in Figure 3 for FMG_{id} commute trivially. \blacktriangleleft

As hinted by the universal property of $\text{FMG}(X)$, given by its elimination rule, the functor FMG is free.

► **Proposition 11.** *FMG is a free functor, and hence the monoidal groupoid $\text{FMG}(X)$ is freely generated by X , for every $X : \mathcal{U}$.*

Proof. We will proceed to fulfil the requirements listed in Definition 6 for a free functor.

- For $X : \mathcal{U}$ and $M : \text{MonGpd}$, a function $\phi_{X,M} : \text{MonGpd}(\text{FMG}(X), M) \rightarrow X \rightarrow M$ is given by $\phi_{X,M}(G) := G \circ \iota$. Then, given a monoidal functor $H : \text{MonGpd}(M, N)$, we have $H \circ \phi_{X,M}(G) \equiv \phi_{X,N}(H \circ G)$, so the diagram in Figure 4a commutes judgmentally (and hence pointwise) and ϕ is natural in M .
- Referring to Remark 9, for $X : \mathcal{U}$ and $M : \text{MonGpd}$, a function $\psi_{X,M} : (X \rightarrow M) \rightarrow \text{MonGpd}(\text{FMG}(X), M)$ is immediately obtained, defining

$$\psi_{X,M}(g) := \text{rec}_{\text{FMG}}(e^M, g, \bullet^M, \alpha^M, \dots) : \text{FMG}(X) \rightarrow M,$$

where $e^M, \bullet^M, \alpha^M, \dots$ are the components of the monoidal structure of M , since the recursor of FMG is a monoidal functor. If $h : Y \rightarrow X$ is a function of types, a monoidal natural isomorphism

$$\theta_\psi : \text{MonFun}_{\text{FMG}(Y), M}(\psi_{X,M}(g) \circ \text{FMG}(h), \psi_{Y,M}(g \circ h))$$

witnessing naturality of ψ in X can be given as follows. The natural transformation between the underlying monoidal functors

$$\theta_\psi : \Pi(w : \text{FMG}(Y)). \psi_{X,M}(g)(\text{FMG}(h)(w)) = \psi_{Y,M}(g \circ h)(w)$$

is defined as $\theta_\psi := \text{ind}_{\text{FMG}}(e', \iota', \bullet', \dots)$, where:

- $e' := \text{refl} : \psi_{X,M}(g)(\text{FMG}(h)(e)) = \psi_{Y,M}(g \circ h)(e)$, as both sides of the equality compute to e^M ;

- $\iota'(y) \equiv \text{refl} : \psi_{X,M}(g)(\text{FMG}(h)(\iota(y))) = \psi_{Y,M}(g \circ h)(\iota(y))$, for every $y : Y$, as both sides of the equality compute to $g(h(y))$;
- $a' \bullet' b' \equiv a' \bullet^M b' : \psi_{X,M}(g)(\text{FMG}(h)(a \bullet b)) = \psi_{Y,M}(g \circ h)(a \bullet b)$ for every $a, b : \text{FMG}(Y)$, $a' : \psi_{X,M}(g)(\text{FMG}(h)(a)) = \psi_{Y,M}(g \circ h)(a)$ and $b' : \psi_{X,M}(g)(\text{FMG}(h)(b)) = \psi_{Y,M}(g \circ h)(b)$, as the equation computes to

$$\psi_{X,M}(g)(\text{FMG}(h)(a)) \bullet^M \psi_{X,M}(g)(\text{FMG}(h)(b)) = \psi_{Y,M}(g \circ h)(a) \bullet^M \psi_{Y,M}(g \circ h)(b);$$

this way, $\theta_\psi(a \bullet b)$ will compute to $\theta_\psi(a) \bullet^M \theta_\psi(b)$;

- the families α' , λ' and ρ' of pathovers are given by naturality of α^M , λ^M and ρ^M , together with the computation rules of rec_{FMG} in (3.1); for example, α' corresponds to a 2-path filling the diagram in Figure 7;
- the families \sphericalangle' and ∇' of 2-pathovers are trivially given, since they correspond to 3-paths in a 1-type.

The 2-paths $(\theta_\psi)_0$ and $(\theta_\psi)_2(a, b)$ corresponding to the diagrams in Figure 3 for $a, b : \text{FMG}(Y)$ are then trivial; hence, a monoidal natural isomorphism making the diagram in Figure 4b commute is provided and $\psi_{X,M}$ is natural in X .

- A homotopy $\theta : \phi_{X,M} \circ \psi_{X,M} \sim \text{id}_{X \rightarrow M}$, for every $X : \mathcal{U}$ and $M : \text{MonGpd}$ is trivially given, since, for every $g : X \rightarrow M$, we have $\phi_{X,M}(\psi_{X,M}(g)) \equiv \psi_{X,M}(g) \circ \iota \equiv g$.
- A monoidal natural isomorphism $\chi : \text{MonFun}_{\text{FMG}(X), M}(\psi_{X,M}(\phi_{X,M}(G)), G)$ for every $X : \mathcal{U}$, $M : \text{MonGpd}$ and $G : \text{MonGpd}(\text{FMG}(X), M)$ is given as follows. The natural transformation between the underlying monoidal functors

$$\chi : \Pi(w : \text{FMG}(X)). \psi_{X,M}(\phi_{X,M}(G))(w) = G(w)$$

can be defined as $\chi := \text{ind}_{\text{FMG}}(e', \iota', \bullet', \dots)$, where:

- $e' \equiv G_0 : \psi_{X,M}(\phi_{X,M}(G))(e) = G(e)$, since the left-hand side of the equality computes to e^M ;
- $\iota'(x) \equiv \text{refl} : \psi_{X,M}(\phi_{X,M}(G))(\iota(x)) = G(\iota(x))$ for $x : X$, as both sides of the equality are judgmentally equal to $\phi_{X,M}(G)(x)$;
- $a' \bullet' b' : \psi_{X,M}(\phi_{X,M}(G))(a \bullet b) = G(a \bullet b)$ is given recursively, for $a, b : \text{FMG}(X)$, $a' : \psi_{X,M}(\phi_{X,M}(G))(a) = G(a)$ and $b' : \psi_{X,M}(\phi_{X,M}(G))(b) = G(b)$, by the concatenation:

$$\begin{aligned} & \psi_{X,M}(\phi_{X,M}(G))(a \bullet b) \\ & \equiv \psi_{X,M}(\phi_{X,M}(G))(a) \bullet^M \psi_{X,M}(\phi_{X,M}(G))(b) \\ & = G(a) \bullet^M G(b) && \text{by } a' \bullet^M b' \\ & = G(a \bullet b) && \text{by } G_2(a, b); \end{aligned}$$

this way, $\chi(a \bullet b)$ will compute to $(\chi(a) \bullet^M \chi(b)) \cdot G_2(a, b)$;

- the families α' , λ' and ρ' of pathovers are given by the computation rules of $\psi_{X,M}$, naturality of α^M , λ^M and ρ^M , and by G_α , G_λ and G_ρ ; Figure 8 shows α' , while the other families are obtained similarly;
- again, the families \sphericalangle' and ∇' of 2-pathovers are trivially given.

With this definition of the underlying homotopy χ , there are trivial paths χ_0 and $\chi_2(a, b)$ corresponding to the diagrams in Figure 3, making χ into a monoidal natural isomorphism. \blacktriangleleft

4 Coherence

This section is devoted to the proof of Theorem 7; from here on, X is assumed to be a 0-type. Coherence will be proved by exhibiting functions $K : \text{FMG}(X) \rightarrow \text{list}(X)$ and $J : \text{list}(X) \rightarrow \text{FMG}(X)$, where the latter is a retraction of the former.

The type $\text{list}(X)$ has a monoidal structure that sees the operation of list concatenation as the monoidal product, which by list-elimination can be proved associative and unital w.r.t. the unit given by the empty list. Families of coherence 2-paths $\triangleleft^{\text{list}}$ and ∇^{list} are provided by the truncation level of $\text{list}(X)$, which is a 0-type (because X is); the ensuing construction $\text{list} : \mathcal{U} \rightarrow \text{MonGpd}$, when restricted to 0-types, satisfies the conditions to be a functor in the sense of Definition 6. As a matter of fact, $\text{FMG}(X)$ and $\text{list}(X)$ as monoidal groupoids can be shown to be in a monoidal equivalence, where the half adjoint monoidal functors are built upon the mentioned functions K and J . We will only make use of the monoidal components of J to prove the retraction; a complete proof of the monoidal equivalence is present in the Coq formalisation included to this paper as supplementary material.

► **Remark 12.** We will use the following notation and conventions. The constructors of $\text{list}(X)$ are the empty list $\text{nil} : \text{list}(X)$ and $:: : X \rightarrow \text{list}(X) \rightarrow \text{list}(X)$ (infix). Concatenation of lists $++$ (infix) is defined by list-elimination on the first argument, i.e. $\text{nil} ++ l_2 \equiv l_2$ and $(x :: l_1) ++ l_2 \equiv x :: (l_1 ++ l_2)$ for every $x : X$ and $l_1, l_2 : \text{list}(X)$. The components of the monoidal structure of $\text{list}(X)$, besides nil and list concatenation, are $\lambda^{\text{list}} : \Pi(l : \text{list}(X)). \text{nil} ++ l = l$, defined pointwise to be the identity path, and α^{list} and ρ^{list} , defined by list-elimination as follows for every $l, l_1, l_2, l_3 : \text{list}(X)$ and $x : X$:

$$\begin{aligned} \alpha_{l_1, l_2, l_3}^{\text{list}} &: (l_1 ++ l_2) ++ l_3 = l_1 ++ (l_2 ++ l_3) & \rho_l^{\text{list}} &: l ++ \text{nil} = l \\ \alpha_{\text{nil}, l_2, l_3}^{\text{list}} &::: \text{refl} & \rho_{\text{nil}}^{\text{list}} &::: \text{refl} \\ \alpha_{x :: l_1, l_2, l_3}^{\text{list}} &::: \text{ap}_{(x :: -)}(\alpha_{l_1, l_2, l_3}^{\text{list}}) & \rho_{x :: l}^{\text{list}} &::: \text{ap}_{(x :: -)}(\rho_l^{\text{list}}). \end{aligned}$$

► **Definition 13.** We define a function $K : \text{FMG}(X) \rightarrow \text{list}(X)$ as

$$K ::= \text{rec}_{\text{FMG}}(\text{nil}, (x \mapsto x :: \text{nil}), ++, \alpha^{\text{list}}, \lambda^{\text{list}}, \rho^{\text{list}}, \triangleleft^{\text{list}}, \nabla^{\text{list}});$$

that is, K is defined to map the monoidal structure of $\text{FMG}(X)$ into that of $\text{list}(X)$.

► **Definition 14.** We define a monoidal functor $J : \text{MonGpd}(\text{list}(X), \text{FMG}(X))$ as follows. The underlying function $J : \text{list}(X) \rightarrow \text{FMG}(X)$ is defined by list-elimination, declaring $J(\text{nil}) ::= e$ and, recursively, $J(x :: l) ::= \iota(x) \bullet J(l)$, for every $x : X$ and $l : \text{list}(X)$. A path $J_0 : e = J(\text{nil})$ is then given by refl , while, given $l_1, l_2 : \text{list}(X)$, a path $J_2(l_1, l_2) : J(l_1) \bullet J(l_2) = J(l_1 ++ l_2)$ can be produced by induction on l_1 :

$$\begin{aligned} J_2(\text{nil}, l_2) &: J(\text{nil}) \bullet J(l_2) \equiv e \bullet J(l_2) = J(l_2) \equiv J(\text{nil} ++ l_2) && \text{by } \lambda_{J(l_2)}, \\ J_2(x :: l_1, l_2) &: J(x :: l) \bullet J(l_2) \equiv (\iota(x) \bullet J(l_1)) \bullet J(l_2) \\ &= \iota(x) \bullet (J(l_1) \bullet J(l_2)) && \text{by } \alpha_{\iota(x), J(l_1), J(l_2)} \\ &= \iota(x) \bullet J(l_1 ++ l_2) && \text{by } \text{refl} \bullet J_2(l_1, l_2) \\ &\equiv J(x :: (l_1 ++ l_2)) \equiv J((x :: l_1) ++ l_2). \end{aligned}$$

The construction of the families of 2-paths J_α , J_λ and J_ρ are shown in Figures 9 to 11. Since $++$ satisfies left unitality judgmentally (Remark 12), we can easily find a 2-path $J_\lambda(l)$ for $l : \text{list}(X)$, as the sought diagram (Figure 10) is trivial. Moreover, we have for every path $p : l_1 = l_2$ in $\text{list}(X)$ and $x : X$, a 2-path

$$\text{ap}_J(\text{ap}_{(x :: -)}(p)) = \text{refl}_{\iota(x)} \bullet \text{ap}_J(p) \tag{4.1}$$

by induction on p . This, together with the coherence diagrams in Figure 1 and naturality of associativity and unitality, allows us to define the families of 2-paths $J_\alpha(l_1, l_2, l_3)$ and $J_\rho(l)$ by list-elimination (on the first argument for J_α), as shown in Figure 9 and Figure 11, respectively.

► **Lemma 15.** *There is a homotopy $\eta : \text{id}_{\text{FMG}(X)} \sim J \circ K$, for K and J given in Definitions 13 and 14.*

Proof. A term $\eta : \Pi(w : \text{FMG}(X)). w = J(K(w))$ is given by $\eta := \text{ind}_{\text{FMG}}(e', \iota', \bullet', \dots)$, where:

- $e' := \text{refl} : e = J(K(e))$, since the right-hand side of the equality computes to e ;
- $\iota'(x) := \rho_{\iota(x)}^{-1} : \iota(x) = J(K(\iota(x)))$ for every $x : X$, the right-hand side of the equality computing to $\iota(x) \bullet e$;
- $a' \bullet' b' : a \bullet b = J(K(a \bullet b))$ for $a, b : \text{FMG}(X)$, $a' : a = J(K(a))$ and $b' : b = J(K(b))$, is defined as the concatenation:

$$\begin{aligned} a \bullet b &= J(K(a)) \bullet J(K(b)) && \text{by } a' \bullet b' \\ &= J(K(a) ++ K(b)) \equiv J(K(a \bullet b)) && \text{by } J_2(K(a), K(b)), \end{aligned}$$

so that $\eta(a \bullet b) \equiv (\eta(a) \bullet \eta(b)) \cdot J_2(K(a), K(b))$;

- α', λ' and ρ' correspond to the diagrams illustrated in Figures 12 to 14 and are proved using the monoidal components J_α, J_λ and J_ρ of J ;
- the families \triangleleft' and \triangleleft' correspond to 3-paths in a 1-type, so they are obtained trivially. ◀

The proof of coherence is then immediately achieved.

Proof of Theorem 7. By Proposition 11, FMG is a free functor. By Lemma 15, $\text{FMG}(X)$ is a retract of $\text{list}(X)$; since X is a 0-type, so is $\text{list}(X)$ and hence $\text{FMG}(X)$, as shown in Figure 15. ◀

5 Discussion

The choice of employing the higher groupoid structure of types to represent categories, where paths take the role of arrows, leads to an important observation concerning the expressivity of the theory: the concept of strictness of a monoidal category cannot be formulated in the framework. Indeed, strictness is not homotopy invariant, and hence it cannot be detected by the theory. For this reason, we use instead a formulation of coherence (Statement 1) concerning a property that is preserved under equivalence of types. Its proof is, then, the presentation of a technique of *normalisation* of monoidal expressions over a set: a term $a : \text{FMG}(X)$ has, as normal form, the term $J(K(a))$, for J and K given in Definitions 13 and 14.

The use of identity types and HITs in HoTT to describe monoidal structures largely simplifies the definition of the free monoidal groupoid, compared to [4, 5]; there, the free monoidal category over a set X is defined via:

- an inductive set of objects, whose terms correspond to the ones produced by the 0-constructors e, ι and \bullet in $\text{FMG}(X)$;
- inductive families of arrows, with identity arrows, $(- \circ -)$, $(- \bullet -)$, $\alpha, \alpha^{-1}, \lambda, \lambda^{-1}, \rho$ and ρ^{-1} as constructors, on which induction is performed when proving coherence; in our implementation, the groupoid structure of identity types takes care of most of the inductive cases, whereas the cases for α, λ and ρ remain present in the application of the elimination principle of $\text{FMG}(X)$;

8:12 Coherence for Monoidal Groupoids in HoTT

- inductive families of equalities between arrows, with a sizeable number of constructors, including: reflexivity, symmetry and transitivity of equality; associativity and unitality of composition; substitution for composition and for the monoidal product; naturality of associativity and unitality of the monoidal product; the interchange law between composition and the monoidal product; composition of associativity and unitality arrows with their inverse; and the coherence diagrams. All but the latter is made redundant in our implementation, as path induction proves everything but the defining diagrams \diamond and ∇ of the monoidal structure.

As a result, the proof of coherence presented in this paper is considerably shorter than the one provided in [4, 5]; this is reflected in the rather compact computer verification (Section 6), which is included in this paper as supplementary material.

Another feature of our approach is the easiness in formulating and proving freeness of FMG (Proposition 11), as the proof is entirely contained in the elimination principle of the HIT itself. Although this proof was not present in [4, 5], our presentation shares the same strategy in defining *ad hoc* a free structure with inductively generated “objects” and “arrows” and normalising the ensuing monoidal expressions to a type of lists (via the functor K in Definition 13), where coherence is immediate, rather than directly show freeness for lists. This turns out to be the easiest option: indeed, for $X : \mathcal{U}$ and $M : \text{MonGpd}$, the reasonable way of producing functions $\phi_{X,M}$ and $\psi_{X,M}$ as in Definition 6 for $F := \text{list}$ would be by defining:

- $\phi_{X,M}(G) := G \circ (- :: \text{nil}) : X \rightarrow M$ for every $G : \text{MonGpd}(\text{list}(X), M)$;
- the underlying function of $\psi_{X,M}(g) : \text{MonGpd}(\text{list}(X), M)$, for $g : X \rightarrow M$, by list-elimination, in a way analogous to J in Definition 14, i.e., declaring $\psi_{X,M}(g)(\text{nil}) := e^M$ and $\psi_{X,M}(g)(x :: l) := g(x) \bullet^M \psi_{X,M}(g)(l)$ for every $x : X$ and $l : \text{list}(X)$.

The term $(\psi_{X,M}(g))_2$, necessary to define a monoidal functor, and similarly $(\psi_{X,M}(g))_\alpha$, $(\psi_{X,M}(g))_\lambda$ and $(\psi_{X,M}(g))_\rho$, are not trivial and require to be also proved by list-elimination. Carrying convoluted proof terms in a proof of freeness of list is cumbersome and in stark contrast to the benefits given by the computation rules of ind_{FMG} , described in Remark 9 and used in Proposition 11. Moreover, in exhibiting a homotopy $\theta_{X,M} : \phi_{X,M} \circ \psi_{X,M} \sim \text{id}_{X \rightarrow M}$, we would be forced to use function extensionality: indeed, for $g : X \rightarrow M$, we would have to provide a term

$$\theta_{X,M}(g) : \phi_{X,M}(\psi_{X,M}(g)) \equiv (x \mapsto g(x) \bullet^M e^M) = g,$$

whereas $\rho^M \circ g : \Pi(x : X).g(x) \bullet^M e^M = g(x)$ only ensures pointwise equality between the two functions. We believe it is worth observing that, contrary to what above, our proof of freeness of FMG does never employ function extensionality.

Alternative definitions for a monoidal functor $K : \text{MonGpd}(\text{FMG}(X), \text{list}(X))$ are possible. Notably, the normalising functor described in [4, 5] could be replicated in our set-up as a monoidal functor having, as underlying function, the composition of

$$\text{ev}_{\text{nil}} : (\text{list}(X) \rightarrow \text{list}(X)) \rightarrow \text{list}(X), \quad \text{ev}_{\text{nil}}(f) := f(\text{nil})$$

after a function $N : \text{FMG}(X) \rightarrow (\text{list}(X) \rightarrow \text{list}(X))$, defined by the elimination principle of FMG, declaring $N(e) := \text{id}_{\text{list}(X)}$, $N(\iota(x)) := (x :: -)$ for every $x : X$, and $N(a \bullet b) := N(b) \circ N(a)$ for every $a, b : \text{FMG}(X)$. The requirements relative to the higher constructors in the inductive definition of N are trivially fulfilled (since e.g. the composition of functions is judgmentally associative) and the composition $\text{ev}_{\text{nil}} \circ N$ can be shown to be a monoidal functor. While this definition is useful to normalise associativity and unitality, it does not

extend to more complex monoidal structures, for example when symmetry is present (see Section 7), as composition of functions is not symmetric; in that case, a functor such as the one presented in Definition 13 is more suitable to the task of normalisation.

6 Formalisation in Coq

The formalisation, included to this paper as supplementary material, has been written using the HoTT library [11, 2] for the Coq proof assistant³ and is structured as follows:

- the categorical framework on monoidal groupoids is included in `monoidalgroupoid.v`; for the definitions of monoidal groupoids, monoidal functors, monoidal natural isomorphisms and free functors, we use classes instead of Σ -types for easy access to their components and for type coercions;
- in `FMG.v` we provide the definition, for a type X , of the HIT $\text{FMG}(X)$ as a private inductive type, specifying the 0-constructors (on which Coq can perform pattern matching) and, separately, the higher constructors as axioms. The elimination principle ind_{FMG} and its computation rules also need to be given as axioms, while the corresponding recursor rec_{FMG} can be derived from ind_{FMG} ; a specific (derived) version of the elimination principle for families of paths in a groupoid is also formalised;
- the proof of Proposition 11 appears in `FMG_free.v`;
- the proof that $\text{list}(X)$ is a 0-type whenever X is a 0-type is included in `lists.v`; this is achieved by means of an “encode-decode” proof [16, 21] and is roughly based on the fact that, for every $x_1, x_2 : X$ and $l_1, l_2 : \text{list}(X)$, there is an equivalence of identity types $(x_1 :: l_1 = x_2 :: l_2) \simeq (x_1 = x_2) \times (l_1 = l_2)$. The same file contains the definition of the monoidal structure of $\text{list}(X)$;
- Theorem 7 is formalised in `FMG_coherence.v`;
- a library of lemmata about path algebra is included in a separate file (`hott_lemmas.v`).

7 Conclusions

The work presented in this paper serves as an example to highlight some of the features of HoTT that can be employed in the context of formalisation of mathematics. Identity types and higher inductive types can be used to give a concrete description of objects satisfying certain universal properties. This result opens the way to the formalisation of similar coherence theorems; for example, definitions analogue to those given in Sections 2 and 3 can be used to describe (free) *symmetric* monoidal groupoids, and symmetric monoidal expressions can be normalised to a HIT of lists with added paths and 2-paths encoding the action of symmetric groups, corresponding to transpositions of adjacent elements in a list and the relations they satisfy. The relevant formalisation in Coq is also present as part of the supplementary material included.

References

- 1 Sten Agerholm, Ilya Beylin, and Peter Dybjer. A comparison of HOL and ALF formalizations of a categorical coherence theorem. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 17–32, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. doi:10.1007/BFb0105394.

³ Commit 68774877142adbd435ea5013c5f201f3ec6ff66a (February 14th, 2020) of the HoTT library; Coq 8.10+alpha.

- 2 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Mike Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT library: A formalization of homotopy type theory in Coq, 2016. [arXiv:1610.04591](https://arxiv.org/abs/1610.04591).
- 3 Benno van den Berg and Richard Garner. Types are weak ω -groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011. doi:10.1112/plms/pdq026.
- 4 Ilya Beylin. *An ALF Proof of Mac Lane’s Coherence Theorem*. Licentiate thesis (revision: 5.26), Department of Computing Science, Chalmers / Göteborg University, 1997.
- 5 Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 47–61, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. doi:10.1007/3-540-61780-9_61.
- 6 Guillaume Brunerie. On the homotopy groups of spheres in homotopy type theory, June 2016. [arXiv:1606.05916](https://arxiv.org/abs/1606.05916).
- 7 Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Tim Baumann, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. Homotopy Type Theory in Agda. URL: <https://github.com/HoTT/HoTT-Agda>.
- 8 Floris van Doorn. On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory, 2018. [arXiv:1808.10690](https://arxiv.org/abs/1808.10690).
- 9 Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. Homotopy Type Theory in Lean. *Lecture Notes in Computer Science*, pages 479–495, 2017. doi:10.1007/978-3-319-66107-0_30.
- 10 Mike Gordon. Introduction to the HOL System. In *1991 International Workshop on the HOL Theorem Proving System and Its Applications*, pages 2–3, August 1991. doi:10.1109/HOL.1991.596265.
- 11 The HoTT library. URL: <https://github.com/HoTT/HoTT>.
- 12 INRIA – The Coq Proof Assistant. URL: <https://coq.inria.fr/>.
- 13 Gregory Maxwell Kelly. On MacLane’s conditions for coherence of natural associativities, commutativities, etc. *Journal of Algebra*, 1(4):397–402, 1964. doi:10.1016/0021-8693(64)90018-3.
- 14 Tom Leinster. *Higher operads, higher categories*, volume 298 of *London Mathematical Society lecture note series*. Cambridge University Press, Cambridge, 2004. doi:10.1017/CB09780511525896.
- 15 Daniel R. Licata and Guillaume Brunerie. A Cubical Approach to Synthetic Homotopy Theory. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 92–103, 2015. doi:10.1109/LICS.2015.19.
- 16 Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS ’13, pages 223–232, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/LICS.2013.28.
- 17 Peter LeFanu Lumsdaine. Weak ω -categories from intensional type theory. *Logical Methods in Computer Science*, Volume 6, Issue 3, September 2010. doi:10.2168/LMCS-6(3:24)2010.
- 18 Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, pages 1–50, June 2019. doi:10.1017/s030500411900015x.
- 19 Saunders Mac Lane. *Categories for the working mathematician*, volume 5 of *Graduate texts in mathematics*. Springer, New York, 2nd ed. edition, 1998.
- 20 Lena Magnusson. *The Implementation of ALF – a Proof Editor based on Martin-Löf’s Monomorphic Type Theory with Explicit Substitution*. PhD Thesis, Göteborg University and Chalmers University of Technology, 1995.
- 21 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.

A Figures in Proofs

Derived Coherence Diagrams in Figure 1

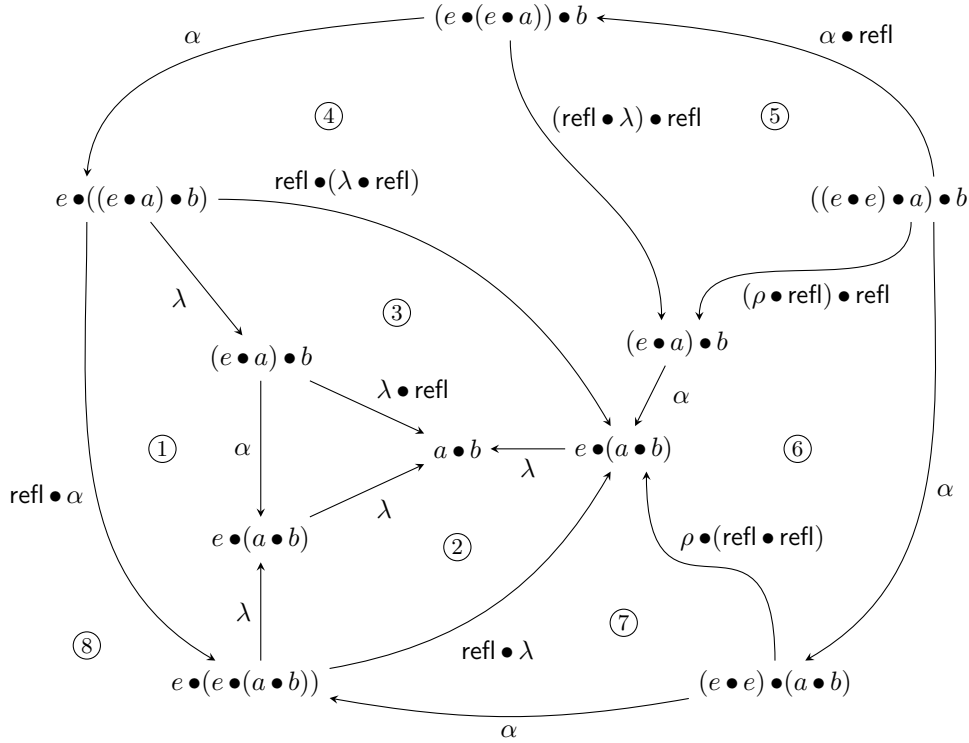


Figure 5 Derivation of the coherence diagram in Figure 1c, here appearing as the unmarked 2-path; the attribute $-^M$ has been omitted for clarity from e , \bullet , α , λ and ρ . The 2-paths (1), (2) and (3) are instances of naturality of λ^M ; (4) and (6) are instances of naturality of α^M ; (5) and (7) are instances of $\nabla^M \bullet^M \text{refl}$ and ∇^M respectively; the outer pentagon (8) is an instance of \diamond^M . The diagram in Figure 1d is obtained similarly.

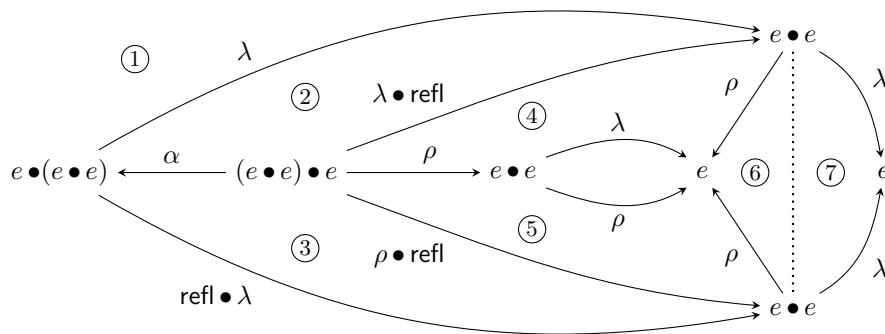


Figure 6 Derivation of the coherence diagram in Figure 1e, here appearing as the unmarked 2-path; again, $-^M$ has been omitted for clarity. The outer square (1) is an instance of naturality of λ^M ; the 2-path (2) is the derived coherence diagram in Figure 1c; (3) is an instance of ∇^M ; (4) and (5) are instances of naturality of ρ^M ; (6) and (7) are trivial.

Figures in the Proof of Proposition 11 (Freeness of FMG)

$$\begin{array}{c}
 (\psi_{X,M}(g)(\text{FMG}(h)(a)) \bullet^M \psi_{X,M}(g)(\text{FMG}(h)(b))) \bullet^M \psi_{X,M}(g)(\text{FMG}(h)(c)) \\
 \vdots \\
 \psi_{X,M}(g)(\text{FMG}(h)((a \bullet b) \bullet c)) \xrightarrow{\theta_\psi((a \bullet b) \bullet c) \equiv (\theta_\psi(a) \bullet^M \theta_\psi(b)) \bullet^M \theta_\psi(c)} \psi_{Y,M}(g \circ h)((a \bullet b) \bullet c) \\
 \text{ap}_{\psi_{X,M}(g) \circ \text{FMG}(h)}(\alpha) \left(\textcircled{1} \right) \alpha^M \quad \textcircled{2} \\
 \psi_{X,M}(g)(\text{FMG}(h)(a \bullet (b \bullet c))) \xrightarrow{\theta_\psi(a) \bullet^M (\theta_\psi(b) \bullet^M \theta_\psi(c)) \equiv \theta_\psi(a \bullet (b \bullet c))} \psi_{Y,M}(g \circ h)(a \bullet (b \bullet c)) \\
 \alpha^M \left(\textcircled{3} \right) \text{ap}_{\psi_{Y,M}(g \circ h)}(\alpha) \\
 \vdots \\
 \psi_{Y,M}(g \circ h)(a) \bullet^M (\psi_{Y,M}(g \circ h)(b) \bullet^M \psi_{Y,M}(g \circ h)(c))
 \end{array}$$

■ **Figure 7** The underlying homotopy θ_ψ in the proof of naturality of $\psi_{X,M}$ in X is achieved via the elimination rules of $\psi_{X,M}$ and FMG; these require certain 2-paths in M to be provided, corresponding to the 1-path constructors of FMG(Y). This figure shows the 2-path α' for associativity. The 2-paths (1) and (3) are given by the computation rules of $\psi_{X,M}$ and FMG(h); (2) is filled by naturality of α^M . The 2-paths λ' and ρ' in M corresponding to the constructors for unitality are proved similarly.

$$\begin{array}{c}
 (\psi_{X,M}(\phi_{X,M}(G))(a) \bullet^M \psi_{X,M}(\phi_{X,M}(G))(b)) \bullet^M \psi_{X,M}(\phi_{X,M}(G))(c) \\
 \vdots \\
 \psi_{X,M}(\phi_{X,M}(G))((a \bullet b) \bullet c) \xrightarrow{\text{ap}_{\psi_{X,M}(\phi_{X,M}(G))}(\alpha)} \psi_{X,M}(\phi_{X,M}(G))(a \bullet (b \bullet c)) \\
 (\chi(a) \bullet^M \chi(b)) \bullet^M \chi(c) \xrightarrow{\alpha^M} \psi_{X,M}(\phi_{X,M}(G))(a \bullet (b \bullet c)) \\
 \textcircled{1} \\
 (G(a) \bullet^M G(b)) \bullet^M G(c) \xrightarrow{\alpha^M} G(a) \bullet^M (G(b) \bullet^M G(c)) \\
 \textcircled{2} \\
 G_2(a, b) \bullet^M \text{refl} \downarrow \quad \downarrow \text{refl} \bullet^M G_2(b, c) \\
 G(a \bullet b) \bullet^M G(c) \quad \textcircled{3} \quad G(a) \bullet^M G(b \bullet c) \\
 G_2(a \bullet b, c) \downarrow \quad \downarrow G_2(a, b \bullet c) \\
 G((a \bullet b) \bullet c) \xrightarrow{\text{ap}_G(\alpha)} G(a \bullet (b \bullet c))
 \end{array}$$

■ **Figure 8** The 2-path in M providing α' in the definition of χ using the elimination principle of FMG(X). The 2-path (1) is given by a computation rule of $\psi_{X,M}$; (2) is an instance of naturality of α^M ; (3) is an instance of G_α . The vertical paths correspond to the ones given by \bullet^M , after application of the interchange law between path concatenation and the action of \bullet^M on paths. The 2-paths for λ' and ρ' are obtained similarly.

Figures relevant to the Proof of Theorem 7 (Coherence for Monoidal Groupoids)

$$\begin{array}{ccc}
(J(\text{nil}) \bullet J(l_2)) \bullet J(l_3) & \xrightarrow{\alpha} & J(\text{nil}) \bullet (J(l_2) \bullet J(l_3)) \\
\vdots & & \vdots \\
(e \bullet J(l_2)) \bullet J(l_3) & \xrightarrow{\textcircled{1}} & e \bullet (J(l_2) \bullet J(l_3)) \\
\lambda \bullet \text{refl} \downarrow & & \downarrow \text{refl} \bullet J_2(l_2, l_3) \\
J(l_2) \bullet J(l_3) & \xrightarrow{\textcircled{2}} & e \bullet J(l_2 ++ l_3) \\
J_2(l_2, l_3) \downarrow & & \downarrow \lambda \\
J(l_2 ++ l_3) & \xrightarrow{\textcircled{3}} & J(l_2 ++ l_3) \\
\vdots & & \vdots \\
J((\text{nil} ++ l_2) ++ l_3) & \xrightarrow{\text{ap}_J(\alpha^{\text{list}}) \equiv \text{refl}} & J(\text{nil} ++ (l_2 ++ l_3))
\end{array}$$

(a) The 2-path $J_\alpha(\text{nil}, l_2, l_3)$ in the inductive definition of J_α . The 2-path (1) is an instance of the additional coherence diagram in Figure 1c; (2) is an instance of naturality of λ ; (3) is trivial.

$$\begin{array}{ccc}
(J(x :: l) \bullet J(l_2)) \bullet J(l_3) & \xrightarrow{\alpha} & J(x :: l) \bullet (J(l_2) \bullet J(l_3)) \\
\vdots & & \vdots \\
((\iota(x) \bullet J(l)) \bullet J(l_2)) \bullet J(l_3) & & (\iota(x) \bullet J(l)) \bullet (J(l_2) \bullet J(l_3)) \\
\alpha \bullet \text{refl} \downarrow & & \downarrow (\text{refl} \bullet \text{refl}) \bullet J_2(l_2, l_3) \\
(\iota(x) \bullet (J(l) \bullet J(l_2))) \bullet J(l_3) & \xrightarrow{\textcircled{1}} & (\iota(x) \bullet J(l)) \bullet (J(l_2) \bullet J(l_3)) \\
(\text{refl} \bullet J_2(l, l_2)) \bullet \text{refl} \downarrow & & \downarrow \alpha \\
(\iota(x) \bullet J(l ++ l_2)) \bullet J(l_3) & \xrightarrow{\text{refl} \bullet \alpha} & (\iota(x) \bullet J(l)) \bullet J(l_2 ++ l_3) \\
\alpha \downarrow & & \downarrow \alpha \\
\iota(x) \bullet (J(l ++ l_2)) \bullet J(l_3) & \xrightarrow{\textcircled{2}} & \iota(x) \bullet (J(l) \bullet J(l_2 ++ l_3)) \\
\text{refl} \bullet J_2(l ++ l_2, l_3) \downarrow & & \downarrow \text{refl} \bullet J_2(l, l_2 ++ l_3) \\
\iota(x) \bullet J((l ++ l_2) ++ l_3) & \xrightarrow{\text{refl} \bullet \text{ap}_J(\alpha^{\text{list}})} & \iota(x) \bullet J(l ++ (l_2 ++ l_3)) \\
\vdots & & \vdots \\
J(x :: ((l ++ l_2) ++ l_3)) & \xrightarrow{\text{ap}_J(\alpha^{\text{list}}) \equiv} & J(x :: (l ++ (l_2 ++ l_3))) \\
\vdots & & \vdots \\
J(((x :: l) ++ l_2) ++ l_3) & \xrightarrow{\text{ap}_J(\text{ap}_{(x :: -)}(\alpha^{\text{list}}))} & J((x :: l) ++ (l_2 ++ l_3))
\end{array}$$

(b) $J_\alpha(x :: l, l_2, l_3)$ in the inductive definition of J_α . The 2-path (1) is an instance of \diamond ; (2) and (3) are instances of naturality of α , where the omitted paths are $\text{refl} \bullet (J_2(l, l_2)) \bullet \text{refl}$ and $\text{refl} \bullet (\text{refl} \bullet J_2(l_2, l_3))$ respectively; (4) is given by $\text{refl}_{\iota(x)} \bullet J_\alpha(l, l_2, l_3)$; (5) is an instance of (4.1).

■ **Figure 9** Construction of the 2-path $J_\alpha(l_1, l_2, l_3)$ by induction on l_1 , after unfolding the definition of J_2 (appearing on the vertical sides) and some path algebra.

$$\begin{array}{ccc}
 e \bullet J(l) & \xrightarrow{\lambda} & J(l) \\
 \text{refl} \bullet \text{refl} \downarrow & & \uparrow \text{ap}_J(\lambda^{\text{list}}) \equiv \text{refl} \\
 e \bullet J(l) & & J(l) \\
 \vdots & & \vdots \\
 J(\text{nil}) \bullet J(l) & \xrightarrow{\lambda} & J(\text{nil} ++ l)
 \end{array}$$

■ **Figure 10** The 2-path $J_\lambda(l)$, after unfolding the definitions of J_0 (left side) and J_2 (bottom side), is trivial.

$$\begin{array}{ccc}
 J(\text{nil}) \bullet e & \xrightarrow{\rho_e} & J(\text{nil}) \\
 \vdots & & \vdots \\
 e \bullet e & & e \\
 \text{refl} \downarrow & & \uparrow \text{ap}_J(\rho^{\text{list}}) \equiv \text{refl} \\
 e \bullet e & & e \\
 \vdots & & \vdots \\
 J(\text{nil}) \bullet J(\text{nil}) & \xrightarrow{\lambda_e} & J(\text{nil} ++ \text{nil})
 \end{array}$$

(a) The 2-path $J_\rho(\text{nil})$ in the inductive definition of J_ρ can be obtained by the additional coherence diagram in Figure 1e.

$$\begin{array}{ccc}
 J(x :: l) \bullet e & \xrightarrow{\rho} & J(x :: l) \\
 \vdots & & \vdots \\
 (\iota(x) \bullet J(l)) \bullet e & & \iota(x) \bullet J(l) \\
 \text{refl} \downarrow & \text{refl} \bullet \rho \curvearrowright & \text{refl} \bullet \text{ap}_J(\rho^{\text{list}}) \\
 \text{①} & \text{②} & \text{③} \\
 (\iota(x) \bullet J(l)) \bullet e & \xrightarrow{\alpha} & \iota(x) \bullet (J(l) \bullet e) & \xrightarrow{\text{refl} \bullet \text{ap}_J(\rho^{\text{list}})} & \iota(x) \bullet J(l) \\
 \downarrow \text{refl} & \downarrow \text{refl} & \downarrow \text{refl} & \downarrow \text{refl} & \downarrow \text{refl} \\
 (\iota(x) \bullet J(l)) \bullet e & \xrightarrow{\alpha} & \iota(x) \bullet (J(l) \bullet e) & \xrightarrow{\text{refl} \bullet J_2(l, \text{nil})} & \iota(x) \bullet J(l ++ \text{nil}) \\
 \vdots & & \vdots & & \vdots \\
 J(x :: l) \bullet J(\text{nil}) & \xrightarrow{J_2(x :: l, \text{nil})} & J(x :: l ++ \text{nil})
 \end{array}$$

(b) The 2-path $J_\rho(x :: l)$ in the inductive definition of J_ρ . The 2-path (1) is an instance of the additional coherence diagram in Figure 1d; (2) is given recursively by $\text{refl}_{\iota(x)} \bullet J_\rho(l)$; (3) is an instance of (4.1).

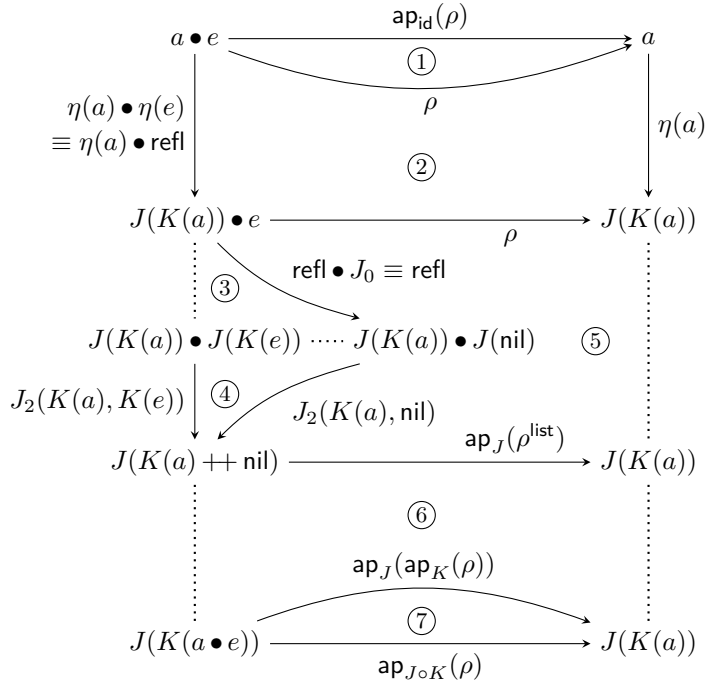
■ **Figure 11** The construction of the 2-path $J_\rho(l)$ by induction on l , after unfolding the definitions of J_0 (left side) and J_2 (bottom side) and some path algebra.

$$\begin{array}{ccc}
(a \bullet b) \bullet c & \xrightarrow{\text{ap}_{\text{id}}(\alpha)} & a \bullet (b \bullet c) \\
\downarrow (\eta(a) \bullet \eta(b)) \bullet \eta(c) & \textcircled{1} \quad \alpha & \downarrow \eta(a) \bullet (\eta(b) \bullet \eta(c)) \\
(J(K(a)) \bullet J(K(b))) \bullet J(K(c)) & \xrightarrow{\alpha} & J(K(a)) \bullet (J(K(b)) \bullet J(K(c))) \\
\downarrow J_2(K(a), K(b)) \bullet \text{refl} & \textcircled{2} & \downarrow \text{refl} \bullet J_2(K(b), K(c)) \\
J(K(a) ++ K(b)) \bullet J(K(c)) & \textcircled{3} & J(K(a)) \bullet J(K(b) ++ K(c)) \\
\downarrow J_2(K(a) ++ K(b), K(c)) & & \downarrow J_2(K(a), K(b) ++ K(c)) \\
J((K(a) ++ K(b)) ++ K(c)) & \xrightarrow{\text{ap}_J(\alpha^{\text{list}})} & J(K(a) ++ (K(b) ++ K(c))) \\
\vdots & \textcircled{4} & \vdots \\
J(K((a \bullet b) \bullet c)) & \xrightarrow{\text{ap}_J(\text{ap}_K(\alpha))} & J(K(a \bullet (b \bullet c))) \\
& \textcircled{5} & \\
& \text{ap}_{J \circ K}(\alpha) &
\end{array}$$

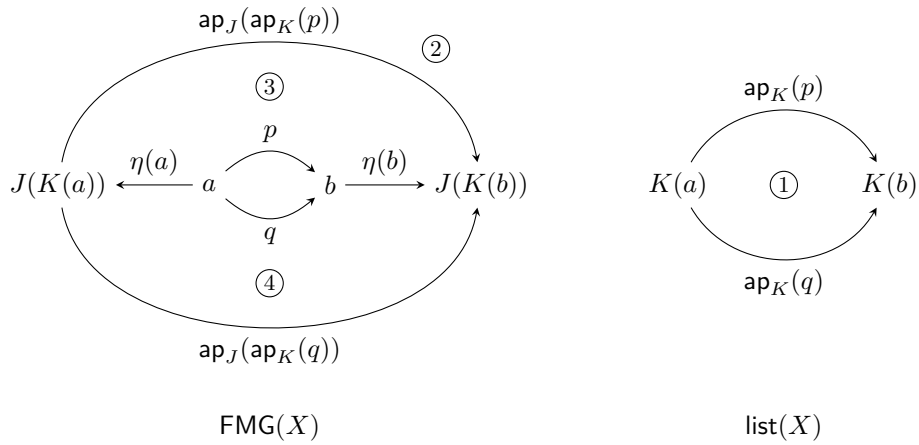
■ **Figure 12** The 2-path α' in the definition of η . The vertical path on the left is equal to $\eta((a \bullet b) \bullet c)$ using the interchange law between path concatenation and the action of \bullet on paths; similarly, the vertical path on the right is equal to $\eta(a \bullet (b \bullet c))$. The 2-paths (1) and (5) are given by path algebra; (2) is an instance of naturality of α ; (3) is an instance of J_α ; (4) is given by a computation rule of K .

$$\begin{array}{ccc}
e \bullet b & \xrightarrow{\text{ap}_{\text{id}}(\lambda)} & b \\
\downarrow \eta(e) \bullet \eta(b) & \textcircled{1} \quad \lambda & \downarrow \eta(b) \\
\equiv \text{refl} \bullet \eta(b) & \textcircled{2} & \\
e \bullet J(K(b)) & \xrightarrow{\lambda} & J(K(b)) \\
\downarrow J_2(\text{nil}, K(b)) & \textcircled{3} & \downarrow \vdots \\
\equiv \lambda & & \\
J(\text{nil} ++ K(b)) \cdots J(K(b)) & \xrightarrow{\text{ap}_J(\lambda^{\text{list}})} & J(K(b)) \\
\vdots & \textcircled{4} & \vdots \\
J(K(e \bullet b)) & \xrightarrow{\text{ap}_J(\text{ap}_K(\lambda))} & J(K(b)) \\
& \textcircled{5} & \\
& \text{ap}_{J \circ K}(\lambda) &
\end{array}$$

■ **Figure 13** The 2-path λ' in the definition of η ; the vertical path on the left is by definition $\eta(e \bullet b)$. The 2-paths (1) and (5) are given by path algebra; (2) is an instance of naturality of λ ; (3) is trivial, as $\lambda_{K(b)}^{\text{list}} \equiv \text{refl}$; (4) is given by a computation rule of K .



■ **Figure 14** The 2-path ρ' in the definition of η ; the vertical path on the left is by definition $\eta(a \bullet e)$. The 2-paths (1) and (7) are given by path algebra; (2) is an instance of naturality of ρ ; (3) and (4) are trivial; (5) is an instance of J_ρ ; (6) is given by a computation rule of K .



■ **Figure 15** Proof of coherence. For any two paths $p, q : a = b$ in $\text{FMG}(X)$, there is a 2-path (1) in $\text{list}(X)$, since this is a 0-type. By functoriality, we obtain a 2-path in $\text{FMG}(X)$ corresponding to the outer diagram (2). The 2-paths (3) and (4) are obtained by path induction (on p and q respectively), yielding a term in $p = q$ corresponding to the unmarked 2-path.

Is Impredicativity Implicitly Implicit?

Stefan Monnier 

Université de Montréal – DIRO, Canada
monnier@iro.umontreal.ca

Nathaniel Bos

McGill University – SOCS, Montréal, Canada
nathaniel.bos@mail.mcgill.ca

Abstract

Of all the threats to the consistency of a type system, such as side effects and recursion, impredicativity is arguably the least understood. In this paper, we try to investigate it using a kind of blackbox reverse-engineering approach to map the landscape. We look at it with a particular focus on its interaction with the notion of *implicit* arguments, also known as *erasable* arguments.

More specifically, we revisit several famous type systems believed to be consistent and which do include some form of impredicativity, and show that they can be refined to equivalent systems where impredicative quantification can be marked as erasable, in a stricter sense than the kind of proof irrelevance notion used for example for **Prop** terms in systems like Coq.

We hope these observations will lead to a better understanding of why and when impredicativity can be sound. As a first step in this direction, we discuss how these results suggest some extensions of existing systems where constraining impredicativity to erasable quantifications might help preserve consistency.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Higher order logic; Software and its engineering → Functional languages

Keywords and phrases Impredicativity, Pure type systems, Inductive types, Erasable arguments, Proof irrelevance, Implicit arguments, Universe polymorphism

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.9

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant N° 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

Acknowledgements We would like to thank Chris League for his comments on earlier drafts of the paper, as well as the reviewers for their careful reading and very helpful feedback.

1 Introduction

Russell introduced the notion of *type* and *predicativity* as a way to stratify our definitions so as to prevent the diagonalization and self-references that lead to logical inconsistencies. This stratification seems sufficient to protect us from such paradoxes, but it does not seem to be absolutely necessary either: systems such as System-F are not predicative yet they are generally believed to be consistent. Some people reject impredicativity outright, and indeed systems like Agda [8] demonstrate that you can go a long way without impredicativity, yet, many popular systems, like Coq [18], do include some limited form of impredicativity. But those limits tend to feel somewhat ad-hoc, making the overall system more complex, with unsatisfying corner cases. For this reason we feel there is still a need to try and better understand what those limits to impredicativity should look like.

Let's disappoint the optimistic reader right away: we won't solve this problem. But during the design of our experimental language Typer [24], we noticed a property shared by several existing impredicative systems, that seemed to link impredicativity and erasability.



© Stefan Monnier and Nathaniel Bos;

licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 9; pp. 9:1–9:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

9:2 Is Impredicativity Implicitly Implicit?

Some mathematicians, such as Carnap [13], have argued that impredicative quantification might be acceptable as long as those arguments are not used in a, we shall say, “significant” way. So in a sense this article investigates whether erasability might be such a notion of “insignificance”.

The two main instances of impredicativity in modern type theory are probably Coq’s `Prop` universe, which is designed to be erasable, and the propositional resizing axiom [27] which allows the use of impredicativity for all *mere propositions*, i.e. types whose inhabitants are all provably equal and hence erasable. For this reason, it is no ground breaking revelation to claim that there is an affinity between impredicativity and erasability, yet it is still unclear to what extent the two belong together nor which particular form of erasability would be the true soulmate of impredicativity.

While Coq and the propositional resizing axiom basically link impredicativity to the concept of erasure usually called *proof irrelevance*, where an argument is deemed erasable if its type has at most one inhabitant, in this article we investigate its connection to a different form of erasability, where an argument is deemed erasable if the function only uses it in type annotations. This is the notion of erasability found in systems like ICC* and EPTS [5, 22].

More specifically, in Section 3, we take various well-known impredicative systems, refine them with annotations of *erasability*, and then show that all impredicatively quantified arguments can be annotated as erasable. In other words, we show that those existing systems already *implicitly* restrict the arguments to their impredicative quantifications to be erasable. This suggests that maybe a good rule of thumb to keep impredicative quantification sound is to make sure its argument is always erasable.

Armed with this proverbial hammer, we then look at the two main limitations of impredicative quantification in existing systems: the restriction we call no-SELIT (which disallows strong elimination of large inductive types) in systems like Coq, and the fact that only the bottom universe can be impredicative. We then propose systems that replace those somewhat ad-hoc restrictions with the arguably less ad-hoc restriction that impredicative quantification is restricted to erasable quantification. The contributions of this work are:

- A proof that in $CC\omega$ all arguments to impredicative functions are erasable.
- A proof that in the CIC resulting from extending $CC\omega$ with inductive types in the impredicative universe, all arguments to impredicative functions and all *large* fields of inductive types are also erasable.
- A new calculus ECIC which lifts the no-SELIT restriction, i.e. it extends CIC with strong elimination of large inductive types.
- A proof that restricting impredicativity to erasable quantifiers does not directly make impredicativity in more than one universe consistent.
- A new calculus $EpCC\omega$ with an impredicative universe polymorphism which allows more powerful forms of impredicativity, such as a Church encoding with strong elimination.
- As needed for some of the above contributions, we sketch an extension of ICC* with both inductive types. While this is straightforward, we do not know of such a system published so far, the closest we found being the one by Bernardo in [6] and Tejsack’s thesis [26].

$$\begin{array}{ll}
(\text{var}) & x, y, t, l \in \mathcal{V} \\
(\text{sort}) & s \in \mathcal{S} \\
(\text{argkind}) & k, c ::= n \mid e \\
(\text{term}) & e, \tau ::= s \mid x \mid (x:\tau_1) \xrightarrow{k} \tau_2 \mid \lambda x:\tau \xrightarrow{k} e \mid e_1 @^k e_2 \\
(\text{context}) & \Gamma ::= \bullet \mid \Gamma, x:\tau \\
\text{primitive reductions:} & (\lambda x:\tau \xrightarrow{k} e_1) @^k e_2 \rightsquigarrow e_1[e_2/x]
\end{array}$$

■ **Figure 1** Syntax and reduction rules of EPTS.

2 Background

Here we present the notion of erasability we use in the rest of the paper.

2.1 Erasable Pure Type Systems

The calculi we use in this paper are erasable pure type systems (EPTS) [22], which are pure type systems (PTS) [4] extended with a notion of erasability. We use a notation that makes it more clear that the erasability is just an annotation like that of colored pure type systems (CPTS) [7] where the color indicates which arguments are ‘n’ormal and which are ‘e’rasable. The syntax of the terms and computation rules are shown in Figure 1.

A specific EPTS is then defined by providing the triplet $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ which defines respectively the sorts, axioms, and rules of this system. The difference with a plain pure type system, is that the annotation on a function or function call has to match the annotation of the function’s type and that the elements of \mathcal{R} have an additional k indicating to which color this rule applies: rules in \mathcal{R} have the form (k, s_1, s_2, s_3) which means that a function of color k taking arguments in universe s_1 to values in universe s_2 itself lives in universe s_3 . For example, we can define an EPTS which defines a version of System-F with erasability as follows:

$$\begin{array}{l}
\mathcal{S} = \{ *, \square \} \\
\mathcal{A} = \{ (*, \square) \} \\
\mathcal{R} = \{ (k, *, *, *), (k, \square, *, *) \mid k \in \{n, e\} \}
\end{array}$$

This version has 4 different abstractions, allowing both System-F’s value abstractions λ and type abstractions Λ to be annotated as either erasable or normal. It is well known that System-F enjoys the phase distinction [9], which means that all types can be erased before evaluating the terms, so we could also define an EPTS equivalent to System-F with only 2 abstractions, using the following rules instead:

$$\mathcal{R} = \{ (n, *, *, *), (e, \square, *, *) \}$$

This is an example of an impredicative calculus where we can make all impredicative abstractions (in this case, those introduced by the rule $(\square, *, *)$ in the PTS) erasable.

Figure 2 shows the typing rules of our EPTS. Compared to a normal CPTS, the only difference is that the typing rule for functions is split into N-LAM and E-LAM where E-LAM includes the additional constraint $x \notin \text{fv}(e^*)$ that enforces the erasability of the argument.

9:4 Is Impredicativity Implicitly Implicit?

$$\begin{array}{c}
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{ (VAR)} \qquad \frac{(s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2} \text{ (SORT)} \\
\\
\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1 \simeq \tau_2}{\Gamma \vdash e : \tau_2} \text{ (CONV)} \\
\\
\frac{\Gamma \vdash \tau_1 : s_1 \quad \Gamma, x:\tau_1 \vdash \tau_2 : s_2 \quad (k, s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash (x:\tau_1) \xrightarrow{k} \tau_2 : s_3} \text{ (PI)} \\
\\
\frac{\Gamma \vdash e_1 : (x:\tau_1) \xrightarrow{k} \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 @^k e_2 : \tau_2[e_2/x]} \text{ (APP)} \qquad \frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x:\tau_1 \xrightarrow{n} e : (x:\tau_1) \xrightarrow{n} \tau_2} \text{ (N-LAM)} \\
\\
\frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x:\tau_1 \vdash e : \tau_2 \quad x \notin \text{fv}(e^*)}{\Gamma \vdash \lambda x:\tau_1 \xrightarrow{e} e : (x:\tau_1) \xrightarrow{e} \tau_2} \text{ (E-LAM)}
\end{array}$$

■ **Figure 2** Typing rules of our EPTS.

In the CONV rule, \simeq stands for the ordinary β -convertibility.

The expression “ e^* ” is the *erasure* of e , where the erasure function $(\cdot)^*$ erases type annotations as well as all erasable arguments:

$$\begin{array}{ll}
s^* & = s \\
x^* & = x \\
((x:\tau_1) \xrightarrow{k} \tau_2)^* & = (x:\tau_1^*) \rightarrow \tau_2^* \\
(\lambda x:\tau \xrightarrow{n} e)^* & = \lambda x \rightarrow e^* \\
(\lambda x:\tau \xrightarrow{e} e)^* & = e^* \\
(e_1 @^n e_2)^* & = e_1^* @^n e_2^* \\
(e_1 @^e e_2)^* & = e_1^*
\end{array}$$

This expresses the fact that erasable arguments do not influence evaluation. The codomain of the erasure function is technically another language with a slightly different syntax, i.e. without erasability nor type annotations, but we will gloss over those details here since for the purpose of this article we only really ever need to know if “ $x \in \text{fv}(e^*)$ ” rather than the specific shape of “ e^* ” itself.

Since the new E-LAM rule is strictly more restrictive than the normal one, it is trivial to show that every EPTS S , just like every CPTS, has a corresponding PTS we note $\lfloor S \rfloor$ where erasability annotations have simply be removed, and that any well-typed term e in the EPTS S has a corresponding well-typed term $\lfloor e \rfloor$ in $\lfloor S \rfloor$. More specifically: $\Gamma \vdash e : \tau$ in the EPTS S implies $\lfloor \Gamma \rfloor \vdash \lfloor e \rfloor : \lfloor \tau \rfloor$ in the PTS $\lfloor S \rfloor$. As a corollary, if the corresponding PTS is consistent, the EPTS is also consistent.

2.2 Kinds of erasability

The claim that arguments to impredicative functions can be erased could be considered as trivial if we consider that Coq’s only impredicative universe is **Prop** and that it is also the universe that gets erased during program extraction.

$$\begin{aligned}
\mathcal{S} &= \{ \text{Prop}; \text{Type}_\ell \mid \ell \in \mathbb{N} \} \\
\mathcal{A} &= \{ (\text{Prop} : \text{Type}_0); (\text{Type}_\ell : \text{Type}_{\ell+1}) \mid \ell \in \mathbb{N} \} \\
\mathcal{R} &= \{ (k, \text{Prop}, s, s) \mid k \in \{\text{n}, \text{e}\}, s \in \mathcal{S} \} \\
&\cup \{ (k, \text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\max(\ell_1, \ell_2)}) \mid k \in \{\text{n}, \text{e}\}, \ell_1, \ell_2 \in \mathbb{N} \} \\
&\cup \{ (\text{e}, \text{Type}_\ell, \text{Prop}, \text{Prop}) \mid \ell \in \mathbb{N} \} \\
&\cup \{ (\text{n}, \text{Type}_\ell, \text{Prop}, \text{Prop}) \mid \ell \in \mathbb{N} \} \leftarrow \text{Rule absent from eCC}\omega \text{ and eCIC}
\end{aligned}$$

■ **Figure 3** Definition of $\text{CC}\omega$ (and its little sibling $\text{eCC}\omega$) as EPTS.

But the kind of erasability we use in this article is different from that offered by Coq’s irrelevance of `Prop`: on the one hand it’s more restrictive since the only thing you can do with an erasable argument in an EPTS is to pass it around until you finally put it inside a type annotation, but on the other it’s more flexible because any argument can be erasable, regardless of its type. For example, let us take the following polymorphic identity function in Coq:

Definition identity (t : Prop) (x : t) := x.

We can see that this function is impredicative since “`t`” can be instantiated with the type of `identity`. Coq’s erasure would erase all uses of this function in terms that do not live in `Prop`, whereas we will concentrate here on the fact that the “`t`” argument is erasable because it is only used in type annotations.

In [2], Abel and Scherer discuss various other subtly different notions of erasure. One of the differences they mention is the difference between internal and external erasure. The rules of our EPTS are different in this respect from those of ICC [21] and ICC*[5]: our `CONV` rule requires convertibility of the fully explicit types (which corresponds to external erasure), whereas ICC and ICC* use a rule where convertibility is checked after erasure (so-called internal erasure):

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1^* \simeq \tau_2^*}{\Gamma \vdash e : \tau_2}$$

We use the weaker rule because it is sufficient for our needs and makes it immediately obvious that every well-typed term e in an EPTS S has a corresponding well-typed term $[e]$ in $[S]$. Our results would carry over to systems with the stronger rule, of course.

3 Erasable impredicativity in `Prop`

In this section we show that the impredicative quantification in the bottom universe `Prop` is almost always erasable and armed with this observation along with some circumstantial evidence, we propose to rely on this property in order to lift the no-SELIT restriction.

3.1 $\text{eCC}\omega$: Erasing impredicative arguments of $\text{CC}\omega$

We will start by showing that impredicative arguments in the calculus of constructions extended with a tower of universes ($\text{CC}\omega$) are always erasable. We use $\text{CC}\omega$, shown in Figure 3, because it is arguably the pure type system that is most closely related to existing systems like Coq. It follows the tradition of having a special impredicative `Prop` universe with a tower of predicative universes named `Typeℓ`. $\max(\ell_1, \ell_2)$ denotes simply the least upper bound of ℓ_1 and ℓ_2 .

9:6 Is Impredicativity Implicitly Implicit?

The calculus $[\text{CC}\omega]$ we get by removing the erasability annotations is sometimes also called $\text{CC}\omega$ in the literature. And indeed the two are equivalent: we can see that any well-typed term e in $[\text{CC}\omega]$ has a corresponding well-typed term $[e]$ in $\text{CC}\omega$ such that $[[e]] = e$ by simply making $[\cdot]$ add \mathfrak{n} annotations everywhere. Our calculus $\text{CC}\omega$ is incidentally almost identical to the ICC^* calculus of Barras and Bernardo [5] (except for the CONV rule, as discussed above).

With respect to impredicativity, the relevant rules in $\text{CC}\omega$ are $(\mathfrak{e}, \text{Type}_\ell, \text{Prop}, \text{Prop})$ and $(\mathfrak{n}, \text{Type}_\ell, \text{Prop}, \text{Prop})$ which allow functions in Prop to take arguments in any Type_ℓ . We will now show that the second rule is redundant:

► **Lemma 1** (Confinement of impredicativity in $\text{CC}\omega$).

In $\text{CC}\omega$, if $\Gamma \vdash x : \tau_x$ and $\Gamma \vdash e : \tau_e$ and $\Gamma \vdash \tau_x : \text{Type}_\ell$ and $\Gamma \vdash \tau_e : \text{Prop}$ then x can only appear in e^* within arguments to impredicative functions, i.e. functions whose return values live in Prop and whose arguments don't.

Proof. By induction on the type derivation of e :

- Given $\tau_e : \text{Prop}$, clearly e is too small to be a type like a sort s or an arrow $(y : \tau_1) \xrightarrow{k} \tau_2$, and it is also too small to be x itself.
- If the derivation uses the CONV rule to convert $e : \tau_e$ to $e : \tau'_e$, we know that τ'_e also has type Prop , by virtue of the type preservation property, so we can use the induction hypothesis on $e : \tau'_e$.
- If e is a function $\lambda y : \tau_y \xrightarrow{k} e_y$, then τ_y does not matter since it is erased from e^* and only occurrences of x in e_y is a concern, and since $\tau_e : \text{Prop}$, we also know that the type of e_y is itself in Prop , hence we can use the induction hypothesis on it.
- If e is an application $e_1 @^k e_2$, as above we can apply the induction hypothesis to e_1 . As for e_2 , there are two cases: either e_1 takes an argument of type $\tau_1 : \text{Prop}$ in which case we can again apply the induction hypothesis, or it takes an argument of type $\tau_1 : \text{Type}_{\ell'}$ in which case we're done. ◀

We call $\mathfrak{eCC}\omega$ the restriction of $\text{CC}\omega$ where all arguments to impredicative functions are erasable, i.e. $(\mathfrak{n}, \text{Type}_\ell, \text{Prop}, \text{Prop})$ is removed, as shown in Figure 3.

► **Theorem 2** (Erasability of impredicative arguments in $\text{CC}\omega$).

$\text{CC}\omega$'s rule $(\mathfrak{n}, \text{Type}_\ell, \text{Prop}, \text{Prop})$ is redundant, that is, for any derivation $\Gamma \vdash e : \tau$ in $\text{CC}\omega$ there is a corresponding derivation $\Gamma' \vdash e' : \tau'$ in $\mathfrak{eCC}\omega$ such that $[[\Gamma \vdash e : \tau]] = [[\Gamma' \vdash e' : \tau']]$.

Proof. By induction on the type derivation of e where we systematically replace \mathfrak{n} with \mathfrak{e} on all functions, arrows, and applications that previously relied on the rule $(\mathfrak{n}, \text{Type}_\ell, \text{Prop}, \text{Prop})$. Since the erasability annotation is only used in the typing rule of λ -abstractions, the proof follows trivially for all cases except this one. For λ -abstractions that had an \mathfrak{n} annotation that we need to convert to \mathfrak{e} , we need to satisfy the additional condition that $x \notin \text{fv}(e^*)$, which follows from Lemma 1: In the absence of the rule $(\mathfrak{n}, \text{Type}_\ell, \text{Prop}, \text{Prop})$, all functions of type $(y : \tau_1) \xrightarrow{k} \tau_2$ where $\tau_2 : \text{Prop}$ and $\tau_1 : \text{Type}_{\ell'}$ are necessarily erasable, so Lemma 1 implies that x can never occur in e'^* . ◀

This shows that the erasability of System-F's impredicative type abstractions can be extended to all of $\text{CC}\omega$'s impredicative abstractions as well.

$$\begin{array}{l}
\text{(index)} \quad i \in \mathbb{N} \\
\text{(term)} \quad e, \tau, a, b, p ::= \dots \mid \text{Ind}(x:\tau)\langle\vec{a}\rangle \\
\quad \quad \quad \quad \quad \quad \quad \quad \mid \text{Con}(i, \tau) \\
\quad \quad \quad \quad \quad \quad \quad \quad \mid \langle\tau_r\rangle\text{Case } e \text{ of } \langle\vec{b}\rangle \\
\quad \quad \quad \quad \quad \quad \quad \quad \mid \text{Fix}_i x : \tau = e \\
\\
\text{primitive reductions:} \quad \langle\tau_r\rangle\text{Case } (\text{Con}(i, \tau) \overrightarrow{\text{@}^k e}) \text{ of } \langle\vec{b}\rangle \rightsquigarrow b_i \overrightarrow{\text{@}^k e} \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{Fix}_i x : \tau = e \rightsquigarrow e[(\text{Fix}_i x : \tau = e)/x]
\end{array}$$

■ **Figure 4** Extension of Figure 1’s EPTS with inductive types.

3.2 eCIC: Erasing impredicative arguments of CIC

We now extend this result to a calculus of inductive constructions (CIC). We reuse $\text{CC}\omega$ as the base language and add inductive types to it. The term CIC has been used to refer to many different systems. Here we use it to refer to a variant of the “original” CIC from 1994, which only had 3 universes, in which we collapsed **Set** and **Prop** into a single universe, which we call **Prop** even though it is not restricted to be proof irrelevant like Coq’s **Prop**; for readers more familiar with Coq, our CIC’s **Prop** is more like Coq’s impredicative **Set**. Note also that our CIC does have a tower of universes, like Coq, but its inductive types only exist in the bottom universe, as was the case in the original CIC, which is why we prefer to call it CIC than $\text{CIC}\omega$.

We mostly follow the presentation of Giménez [16] for the syntax of inductive types but we extend its rules according to the presentation of Werner [29] which adds a strong elimination, i.e. the ability to compute a type by case analysis on an inductive type, which is needed for many proofs, even simple ones. The syntax of terms and the computational rules of inductive types are shown in Figure 4. Together with the rules of Figure 3 they define CIC (and its little sibling eCIC).

$\text{Ind}(x:\tau)\langle\vec{a}\rangle$ is a (potentially indexed) inductive type which itself has type τ and whose i^{th} constructor has type a_i , where we use the vector notation \vec{a} to represent a sequence of terms $a_0 \dots a_n$. $\text{Con}(i, \tau)$ denotes the i^{th} constructor of the inductive type e . $\langle\tau_r\rangle\text{Case } e \text{ of } \langle\vec{b}\rangle$ is a case analysis of the term e which should be an object of inductive type; it will dispatch to the corresponding branch b_i if e was built with the i^{th} constructor of the inductive type; τ_r describes the return type of the case expression. Finally $\text{Fix}_i x : \tau = e$ is a recursive function x of type τ , defined by structural induction on its i^{th} argument (the reduction rule shown above is naive, but the details do not affect us here).

We must of course also extend the definition of our erasure function to handle those additional terms:

$$\begin{array}{l}
\text{Ind}(x:\tau)\langle\vec{a}\rangle^* \quad = \quad \text{Ind}(x)\langle\vec{a}^*\rangle \\
\text{Con}(i, \tau)^* \quad = \quad \text{Con}(i) \\
\langle\tau_r\rangle\text{Case } e \text{ of } \langle\vec{b}\rangle^* \quad = \quad \text{Case } e^* \text{ of } \langle\vec{b}^*\rangle \\
(\text{Fix}_i x : \tau = e)^* \quad = \quad \text{Fix } x = e^*
\end{array}$$

While these new terms may appear not to take erasability into account, this is only because the erasability of the fields of those inductive types is introduced by the erasability annotations on the formal arguments of \vec{a} which need to match those of \vec{b} : they really do let you specify the erasability of each field; and every field, whether erasable or not, is available within the corresponding **Case** branch but those marked as erasable in the **Ind** definition will accordingly only be available as erasable within **Case**.

9:8 Is Impredicativity Implicitly Implicit?

$$\begin{array}{c}
\frac{\Gamma \vdash \tau : s \quad \forall i. \quad \Gamma, x:\tau \vdash a_i : \mathbf{Prop} \quad x \vdash a_i \text{ con}}{\Gamma \vdash \mathbf{Ind}(x:\tau)\langle \vec{a} \rangle : \tau} \\
\\
\frac{\tau = \mathbf{Ind}(x:\tau')\langle \vec{a} \rangle \quad \Gamma \vdash \tau : \tau'}{\Gamma \vdash \mathbf{Con}(i, \tau) : a_i[\tau/x]} \quad \frac{\forall i. \quad \Gamma \vdash \tau_i : \mathbf{Prop}}{\Gamma \vdash \vec{\tau} \text{ small}} \\
\\
\frac{\Gamma \vdash e : \tau_I \xrightarrow{\vec{a}} \quad \tau_I = \mathbf{Ind}(x:(z:\tau_z) \xrightarrow{\vec{c}} \mathbf{Prop})\langle \vec{a} \rangle \quad \Gamma \vdash \tau_r : (z:\tau_z) \xrightarrow{\vec{c}} (_:\tau_I \xrightarrow{\vec{a}} z) \xrightarrow{\vec{c}} s \\
\forall i. \quad a_i = (y:\tau_y) \xrightarrow{\vec{c}} x \xrightarrow{\vec{c}} \tau'_y \quad s = \mathbf{Prop} \vee \Gamma \vdash \vec{\tau}_y \text{ small} \\
\forall i. \quad \Gamma \vdash b_i : (y:\tau_y[\tau_I/x]) \xrightarrow{\vec{c}} (\tau_r \xrightarrow{\vec{c}} \tau'_y \xrightarrow{\vec{c}} \mathbf{Con}(i, \tau_I) \xrightarrow{\vec{c}} y))}{\Gamma \vdash \langle \tau_r \rangle \mathbf{Case} \text{ e of } \langle \vec{b} \rangle : \tau_r \xrightarrow{\vec{c}} \tau'_y \xrightarrow{\vec{c}} e} \\
\\
\frac{\Gamma, x_f:\tau \vdash e : \tau \quad e = \lambda y: _ \xrightarrow{\vec{c}} \lambda x_i: _ \xrightarrow{\vec{c}} e_b \quad i = |y| \quad x_f; i; x_i; \emptyset \vdash e_b \text{ term}}{\Gamma \vdash \mathbf{Fix}_i x_f : \tau = e : \tau}
\end{array}$$

■ **Figure 5** Typing rules of inductive types.

Auxiliary judgments: $\Gamma \vdash \vec{\tau} \text{ small}$ checks that the fields $\vec{\tau}$ are all in \mathbf{Prop} .

$x \vdash a_i \text{ con}$ checks that a is strictly positive in x .

$x_f; i; x_i; \emptyset \vdash e_b \text{ term}$ makes sure all recursive calls use structurally decreasing arguments.

Figure 5 shows the typing rules corresponding to each of those four new constructs. Those typing rules are pretty intricate, if not downright scary, and most of the details do not directly affect our argument, so the casual reader may prefer to skip them. We use $_$ at a few places where the actual element does not matter enough to give it a name. The notation $f \xrightarrow{\vec{a}} e$ denotes a curried application with multiple arguments $f \xrightarrow{\vec{a}} e_1 \dots \xrightarrow{\vec{a}} e_n$, and similarly $\lambda x:\tau \xrightarrow{\vec{c}} e$ denotes a curried function of multiple arguments $\lambda x_1:\tau_1 \xrightarrow{\vec{c}_1} \dots \lambda x_n:\tau_n \xrightarrow{\vec{c}_n} e$ and $(x:\tau) \xrightarrow{\vec{c}} e$ denotes the type of such a function $(x_1:\tau_1) \xrightarrow{\vec{c}_1} \dots (x_n:\tau_n) \xrightarrow{\vec{c}_n} e$.

The rules are very similar to those used by Giménez in [16] because they are largely unaffected by the erasability annotations. The only exception is for \mathbf{Case} where we have to make sure that the various erasability annotations match each other, e.g. the vector \vec{c} of erasability annotations placed on a given constructor a_i must match the erasability annotations of the arguments expected by the corresponding branch b_i . Two important details are worth pointing out:

- In the rule for \mathbf{Ind} the type of constructors is restricted to be in \mathbf{Prop} : just like in the original CIC we only allow inductive types in our bottom universe, contrary to what systems like Coq [18] and UTT [20] allow.
- In the \mathbf{Case} rule, the hypotheses $s = \mathbf{Prop} \vee \Gamma \vdash \vec{\tau}_y \text{ small}$ ensure that when the result of the case analysis is not in \mathbf{Prop} , i.e. when this is a form of strong elimination, the inductive type must be **small**, meaning that all its fields must be in \mathbf{Prop} . This “no-SELIT” restriction is taken from Werner [29], with a slightly different presentation because he chose to split the \mathbf{Case} rule into two: one for weak elimination and one for strong elimination.

We do not show the definition of the $x \vdash e$ `con` judgment which ensures that e has the appropriate shape for an inductive constructor, including the strict positivity, nor that of the $x_f; i; x_i; \nu \vdash e$ `term` judgment which ensures that recursive calls are made on structurally smaller terms. Their definition is not affected by the presence of erasability annotations and does not impact our work here.

To show that the $(n, \text{Type}_\ell, \text{Prop}, \text{Prop})$ rule of non-erasable impredicativity is still redundant in this new system, we proceed in the same way:

► **Lemma 3** (Confinement of impredicativity in CIC).

In CIC, if $\Gamma \vdash x : \tau_x$ and $\Gamma \vdash e : \tau_e$ and $\Gamma \vdash \tau_x : \text{Type}_\ell$ and $\Gamma \vdash \tau_e : \text{Prop}$ then x can only appear in e^* within arguments to impredicative functions, i.e. functions whose return values live in `Prop` and whose arguments don't.

Proof. The proof stays the same as for $\text{CC}\omega$, with the following additional cases:

- Given $\tau_e : \text{Prop}$, clearly e is too small to be a type like $\text{Ind}(x:\tau)\langle\vec{a}\rangle$.
- If e is of the form $\text{Con}(i, \tau)$, since τ is erased, the erasure is always closed.
- If e is of the form $\text{Fix}_i x : \tau = e'$, then τ does not matter because it's erased, and we can invoke the inductive hypothesis on e' .
- If e is of the form $\langle\tau_r\rangle\text{Case } e' \text{ of } \langle\vec{b}\rangle$, then τ_r does not matter because it is erased. Furthermore, we can invoke the inductive hypothesis on e' since we know that e' lives in `Prop`, like all our inductive types. Finally since the hypothesis tells us that e lives in `Prop`, all branches b_i must as well, hence we can also invoke the induction hypothesis on every b_i . ◀

We call `eCIC` the restriction of CIC where all arguments to impredicative functions and all large fields of inductive definitions are erasable, i.e. $(n, \text{Type}_\ell, \text{Prop}, \text{Prop})$ is removed.

► **Theorem 4** (Erasability of impredicative arguments in CIC).

CIC's rule $(n, \text{Type}_\ell, \text{Prop}, \text{Prop})$ is redundant, that is, for any derivation $\Gamma \vdash e : \tau$ in CIC there is a corresponding derivation $\Gamma' \vdash e' : \tau'$ in `eCIC` such that $[\Gamma \vdash e : \tau] = [\Gamma' \vdash e' : \tau']$

Proof. As before, by induction on the type derivation of e where we systematically replace `n` with `e` on all functions, arrows, and applications that previously relied on the rule $(n, \text{Type}_\ell, \text{Prop}, \text{Prop})$. The interesting new case is when e is of the form $\langle\tau_r\rangle\text{Case } e' \text{ of } \langle\vec{b}\rangle$: as mentioned, the vector \vec{c} of erasability annotations placed on a given constructor a_i must match the erasability annotations of the arguments expected by the corresponding branch b_i . Since our inductive types all live in `Prop`, it means all fields that live in higher universes have been annotated as erasable. But that in turns means that all corresponding arguments to the branches b_i should also be annotated as erasable. When s is `Prop` (i.e. a weak elimination), this is the case because all arguments of higher universe for functions in `Prop` can only be annotated as erasable. And when s is a higher universe the property is also verified because the $\Gamma \vdash \vec{\tau}_y$ small constraint imposes that none of the arguments are in higher universes so they don't use the $(n, \text{Type}_\ell, \text{Prop}, \text{Prop})$ rule. ◀

This shows that the erasability of System-F's impredicative type abstractions can be extended not only to all of $\text{CC}\omega$'s impredicative abstractions but also to CIC's impredicative abstractions and impredicative inductive types.

3.3 ECIC: Strong elimination of large inductive types

The reason behind the $\Gamma \vdash e$ small special constraint on strong eliminations of CIC in Figure 5 is pretty straightforward: without this restriction, we could use an inductive type such as the following to “smuggle” a value of universe Type_ℓ in a box of universe `Prop`:

9:10 Is Impredicativity Implicitly Implicit?

$$\begin{aligned}
\mathcal{R} = & \{ (k, \text{Prop}, s, s) \mid k \in \{\mathbf{n}, \mathbf{e}\}, s \in \mathcal{S} \} \\
& \cup \{ (k, \text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\max(\ell_1, \ell_2)}) \mid k \in \{\mathbf{n}, \mathbf{e}\}, \ell_1, \ell_2 \in \mathbb{N} \} \\
& \cup \{ (e, \text{Type}_{\ell}, \text{Prop}, \text{Prop}) \mid \ell \in \mathbb{N} \}
\end{aligned}$$

$$\frac{\Gamma \vdash e : \tau_I \overset{\rightarrow}{@^k p} \quad \tau_I = \text{Ind}(x : (z : \tau_z) \overset{k}{\rightarrow} \text{Prop}) \langle \vec{a} \rangle \quad \Gamma \vdash \tau_r : (z : \tau_z) \overset{k}{\rightarrow} (_ : \tau_I \overset{\rightarrow}{@^k z}) \overset{n}{\rightarrow} s}{\forall i. \quad a_i = (y : \tau_y) \overset{c}{\rightarrow} x \overset{\rightarrow}{@^k p'} \quad \Gamma \vdash b_i : (y : \tau_y[\tau_I/x]) \overset{c}{\rightarrow} (\tau_r \overset{\rightarrow}{@^k p'} \overset{n}{@} (\text{Con}(i, \tau_I) \overset{c}{@} y))}{\Gamma \vdash \langle \tau_r \rangle \text{Case } e \text{ of } \langle \vec{b} \rangle : \tau_r \overset{\rightarrow}{@^k p} \overset{n}{@} e}$$

■ **Figure 6** Rules of the ECIC system. The rest is unchanged from eCIC, Figures 1, 2, 4, and 5.

```

Inductive Box (t : Type): Prop := box : t -> Box.
Definition unbox (t : Type) (x : Box t) := match x with
| box x' => x'
end.

```

Note that such a box (a large inductive type) is perfectly valid in CIC, but the $\Gamma \vdash e$ small constraint rejects the `unbox` definition (which uses a strong elimination). If we remove the $\Gamma \vdash e$ small constraint, the effect of such a `box/unbox` pair would be to lower any value of a higher universe to the `Prop` universe and would hence defeat the purpose of the stratification introduced by the tower of universes. This was first shown to be inconsistent in [11].

This restriction makes the system more complex since elimination is allowed from any inductive type to any universe except for the one special case of strong elimination of large inductive types (SELIT). It also significantly weakens the system. For example, in Coq with the `--impredicative-set` option, we can define a large inductive type like:

```

Inductive Ω : Set :=
| int    : Ω
| arrow  : Ω -> Ω -> Ω
| all    : forall k:Set, (k -> Ω) -> Ω.

```

which could be used for example to represent the types of some object language. But we cannot prove properties such as the following variant of Leibniz equality (which we needed in the proof of soundness of our Swiss coercion [23]):

```

forall k1 k2 f1 f2 p,
  all k1 f1 = all k2 f2 -> p k1 f1 -> p k2 f2.

```

In practice, this important restriction significantly reduces the applicability of large inductive types (which partly explains why Coq does not allow them in `Set` any more by default).

While the $\Gamma \vdash e$ small constraint was added to avoid an inconsistency, this same $\Gamma \vdash e$ small is also the key to making our proof of erasability of impredicative arguments work for CIC: it is the detail which makes it possible to mark all the large fields of impredicative inductive definitions as erasable, as we saw in the previous section. This might be a coincidence, of course, yet it suggests a close alignment between the needs of consistency and the need to keep impredicative elements erasable.

Figure 6 shows a refinement of eCIC we call ECIC whose `Case` rule does not have the $\Gamma \vdash e$ small constraint. ECIC is more elegant and regular than CIC thanks to the absence of this special corner case, and it allows typing more terms than eCIC and hence CIC. For instance in ECIC we can define the above Ω inductive type with an erasable k and then prove the mentioned property (with k_1 and k_2 marked as erasable).

Note also that the lack of an $(n, \text{Type}_\ell, \text{Prop}, \text{Prop})$ rule, means we cannot define a `box` as above in this system; instead we are limited to making its content erasable. This in turn prevents us from defining `unbox` since the `x'` would now be erasable so it cannot be returned as-is from the elimination form. In other words, forcing impredicative fields to be erasable also avoids this source of inconsistency usually avoided with the $\Gamma \vdash e$ `small` constraint. Based on this circumstantial evidence, we venture to state the following:

► **Conjecture 5.** *The ECIC system is consistent.*

3.4 SELIT for Coq's proof-irrelevant Prop

The `Prop` universe used in the previous section corresponds to Coq's impredicative `Set` universe, which is disabled by default. Coq's impredicative `Prop` universe is similar except it is designed to be proof-irrelevant. This property is used in two ways: to reflect this property in the system via an axiom and to erase all `Prop` terms when *extracting* a program from a proof. This proof-irrelevance property is enforced by two constraints imposed on the strong elimination of those inductive types that live in `Prop`: first, they have to have a single constructor and second, all fields must live in the `Prop` universe. The first constraint makes sure there is no run-time dispatch based on an erased value, while the second guarantees that the only data we can extract from an erased value is itself erased.

The second constraint is the no-SELIT constraint. So the Conjecture 5 suggests we could relax this restriction and allow strong elimination on any `Prop` type with a single constructor if the fields that do not live in `Prop` are erasable. From the point of view of extraction, we could even relax this further to allow strong elimination on any `Prop` type with a single constructor, and simply treat all the values so extracted as erased.

3.5 eCoq: Erasing impredicativity in Coq and UTT

As noted in Section 3.2, we were careful to restrict our inductive types to live in `Prop`. This was no accident: we rely on this property in the confinement lemma used to show the erasability of all impredicative arguments in CIC. Indeed, confinement does not hold if we can do a case analysis on an inductive type that lives in `Typeℓ` and return a value in `Prop`.

Systems such as Coq and UTT [20] allow impredicative definitions in `Prop`, inductive types in higher universes, and elimination from those inductive types to `Prop`. These systems are hence examples of impredicativity which is not straightforwardly erasable like it is in the systems seen so far. Here is an example of code which relies on this possibility:

```
Inductive List (α : Type0) : Type0 := nil | cons (v : α) (vs : List t).
```

```
Definition ifnil (ts : List Prop) (t : Prop) (x y : t) :=
  match ts with
  | nil => x
  | cons _ _ => y.
```

In Coq, `ifnil` lives in `Prop` because its return value is in `Prop`. If we extend Coq with erasability annotations, the argument “`t`” could be marked as erasable since it only appears in type annotations, but not the other three arguments. To determine in which universe it rests, we would use the rules $(n, \text{Prop}, \text{Prop}, \text{Prop})$ for the last two arguments and $(e, \text{Type}_\ell, \text{Prop}, \text{Prop})$ for the second argument. Those rules obey the principle that impredicativity is restricted to erasable arguments. But for the first argument, we need the rule $(n, \text{Type}_\ell, \text{Prop}, \text{Prop})$ which does not obey this principle.

9:12 Is Impredicativity Implicitly Implicit?

$$\begin{aligned}
\mathcal{R} = & \{ (k, \text{Prop}, s, s) \mid k \in \{\mathbf{n}, \mathbf{e}\}, s \in \mathcal{S} \} \\
& \cup \{ (e, \text{Type}_{\ell}, \text{Prop}, \text{Prop}) \mid \ell \in \mathbb{N} \} \\
& \cup \{ (n, \text{Type}_{\ell}, \text{Prop}, \text{Type}_{\ell}) \mid \ell \in \mathbb{N} \} \\
& \cup \{ (k, \text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\max(\ell_1, \ell_2)}) \mid k \in \{\mathbf{n}, \mathbf{e}\}, \ell_1, \ell_2 \in \mathbb{N} \}
\end{aligned}$$

$$\frac{\Gamma \vdash \tau : s \quad \forall i. \quad \Gamma, x : \tau \vdash a_i : s' \quad x \vdash a_i \text{ con}}{\Gamma \vdash \text{Ind}(x : \tau) \langle \vec{a} \rangle : \tau}$$

$$\frac{\Gamma \vdash e : \tau_I \xrightarrow{\text{@}^k p} \quad \tau_I = \text{Ind}(x : (z : \tau_z) \xrightarrow{k} s') \langle \vec{a} \rangle \quad \Gamma \vdash \tau_r : (z : \tau_z) \xrightarrow{k} (_ : \tau_I \xrightarrow{\text{@}^k z}) \xrightarrow{n} s}{\forall i. \quad a_i = (y : \tau_y) \xrightarrow{c} x \xrightarrow{\text{@}^k p'} \quad \Gamma \vdash b_i : (y : \tau_y [\tau_I / x]) \xrightarrow{c} (\tau_r \xrightarrow{\text{@}^k p'} \text{@}^n (\text{Con}(i, \tau_I) \xrightarrow{c} y)))}{\Gamma \vdash \langle \tau_r \rangle \text{Case } e \text{ of } \langle \vec{b} \rangle : \tau_r \xrightarrow{\text{@}^k p} \text{@}^n e}$$

■ **Figure 7** Rules of the eCoq system.

If we want to obey the principle, we could replace this last rule with the predicative rule $(\mathbf{n}, \text{Type}_{\ell}, \text{Prop}, \text{Type}_{\ell})$ instead. Figure 7 shows the important rules of such a system we call eCoq. With such a system, we would have to adjust the above example in one of two ways:

- Live with the fact that `ifnil` will now live in `Type0` rather than in `Prop`. Experience with Agda and other systems suggests that most code does not rely on impredicativity, so in practice this first approach should be applicable in most cases.
- Mark the non-`Prop` parts of “`ts`” as erasable so that it can live in `Prop`. Concretely, it means using a new type we could call `eList`, which is like `List` except that the “`v`” field of the “`cons`” constructor is marked as erasable, to allow those “thinner” lists to live in `Prop`.

We call the second approach *thinning*. It replaces inductive objects from a higher universe with similar objects that fit in `Prop` by marking the non-`Prop` parts of it as erasable or by replacing them with similarly “thinned” elements.

It is still unclear whether any valid typing derivation in a system like Coq can have a corresponding typing derivation in eCoq, that is, whether we can do away with the $(\mathbf{n}, \text{Type}_{\ell}, \text{Prop}, \text{Prop})$ rule because we can always change the source code as described above.

4 Universe-agnostic impredicativity

CC ω accepts impredicative definitions only in the bottom universe, `Prop`, just like in most known consistent type systems that support impredicative definitions (one counter example being arguably the $\lambda\text{PRED}\omega^+$ presented in [14]). This is a direct consequence of various paradoxes formalized in systems which allow impredicative definitions in more than one universe [17, 12, 19]. In this section we investigate the use of erasability constraints in order to lift this restriction and thus allow impredicative definitions in higher universes as well.

4.1 λeU^- : Erasing impredicative arguments in λU^-

The last two papers referenced above showed a paradox in the system λU^- which is F_{ω} extended with one extra rule. It can be defined as an EPTS as follows:

$$\begin{aligned}
\mathcal{S} &= \{ *, \square, \Delta \} \\
\mathcal{A} &= \{ (*, \square), (\square, \Delta) \} \\
\mathcal{R} &= \{ (k, *, *, *), (k, \square, *, *), (k, \square, \square, \square), (k, \Delta, \square, \square) \mid k \in \{\mathbf{n}, \mathbf{e}\} \}
\end{aligned}$$

$$\begin{aligned}
\mathcal{U} &= \Pi \mathcal{X} : \square. ((\wp \wp \mathcal{X} \rightarrow \mathcal{X}) \rightarrow \wp \wp \mathcal{X}) \\
\tau t &= \Lambda \mathcal{X} : \square. \lambda f : (\wp \wp \mathcal{X} \rightarrow \mathcal{X}). \lambda p : \wp \mathcal{X}. (t \lambda x : \mathcal{U}. (p (f (\{x \mathcal{X}\} f)))) \\
\sigma s &= (\{s \mathcal{U}\} \lambda t : \wp \wp \mathcal{U}. \tau t) \\
\Delta &= \lambda y : \mathcal{U}. \neg \forall p : \wp \mathcal{U}. [(\sigma y p) \Rightarrow (p \tau \sigma y)] \\
\Omega &= \tau \lambda p : \wp \mathcal{U}. \forall x : \mathcal{U}. [(\sigma x p) \Rightarrow (p x)]
\end{aligned}$$

$$\begin{aligned}
& [\text{suppose } 0 : \forall p : \wp \mathcal{U}. [\forall x : \mathcal{U}. [(\sigma x p) \Rightarrow (p x)] \Rightarrow (p \Omega)]. \\
& \quad [\langle 0 \Delta \rangle \text{ let } x : \mathcal{U}. \\
& \quad \quad \text{suppose } 2 : (\sigma x \Delta). \\
& \quad \quad \text{suppose } 3 : (\forall p : \wp \mathcal{U}. [(\sigma x p) \Rightarrow (p \tau \sigma x)]). \\
& \quad \quad [\langle 3 \Delta \rangle 2] \text{ let } p : \wp \mathcal{U}. \langle 3 \lambda y : \mathcal{U}. (p \tau \sigma y) \rangle] \\
& \quad \text{let } p : \wp \mathcal{U}. \langle 0 \lambda y : \mathcal{U}. (p \tau \sigma y) \rangle] \\
& \quad \text{let } p : \wp \mathcal{U}. \\
& \quad \text{suppose } 1 : \forall x : \mathcal{U}. [(\sigma x p) \Rightarrow (p x)]. \\
& \quad [\langle 1 \Omega \rangle \text{ let } x : \mathcal{U}. \langle 1 \tau \sigma x \rangle]
\end{aligned}$$

■ **Figure 8** Hurken's paradox.

Two of the four pairs of rules are impredicative: $(k, \square, *, *)$ and $(k, \Delta, \square, \square)$. The first is generally considered harmless since $*$ is the bottom universe and hence corresponds to **Prop** in $CC\omega$. The new one is $(k, \Delta, \square, \square)$ which introduces impredicativity in the second universe, \square . Following the same idea as in the previous section where we defined **ECIC** to rely on erasability to avoid inconsistency, we could thus define a new λeU^- calculus that only allows the use of impredicativity with erasable abstractions:

$$\mathcal{R} = \{ (k, *, *, *), (e, \square, *, *), (k, \square, \square, \square), (e, \Delta, \square, \square) \quad | \quad k \in \{n, e\} \}$$

Alas, this does not help:

► **Theorem 6.** λeU^- is not consistent.

Proof. The proof is the same as the proof of inconsistency of λU^- shown by Hurkens in [19]. Figure 8 shows Hurken's original proof, using the same notation he used in his paper. To show that the proof also applies to λeU^- , we need to make sure that all impredicative abstractions can be annotated as erasable. For that, it suffices to know that the integers are variable names, the impredicative abstraction in $*$ is introduced by **let**, the corresponding application is denoted with $\langle e_1 e_2 \rangle$, the impredicative abstraction in \square is introduced by Λ , and the corresponding application is denoted with $\{e_1 e_2\}$: by inspection we can see that all the arguments introduced by impredicative abstractions are exclusively used either in type annotations or in arguments to other impredicative functions. ◀

This demonstrates that, even though the notion of erasability we use here has shown strong affinities with consistent uses of impredicativity, it is not in general sufficient to tame the excesses of impredicativity.

4.2 Inductive types: Impredicative and universe polymorphic?

While paradoxes like Hurkens's suggest that it is impossible to have impredicative definitions in more than one universe without losing consistency, inductive definitions suggest otherwise.

9:14 Is Impredicativity Implicitly Implicit?

$$\begin{aligned}
 (\text{level}) \quad \ell & ::= 0 \mid \mathfrak{s} \ell \mid l \mid \ell_1 \sqcup \ell_2 \\
 \mathcal{S} & = \{ \text{UL}; \text{Type}_\ell; \text{Type}_\omega \} \\
 \mathcal{A} & = \{ (\text{Level} : \text{UL}); (\text{Type}_\ell : \text{Type}_{(\mathfrak{s} \ell)}) \} \\
 \mathcal{R} & = \{ (k, \text{UL}, \text{Type}_\ell, \text{Type}_\omega) \mid k \in \{\mathfrak{n}, \mathfrak{e}\} \} \\
 & \cup \{ (k, \text{UL}, \text{Type}_\omega, \text{Type}_\omega) \mid k \in \{\mathfrak{n}, \mathfrak{e}\} \} \\
 & \cup \{ (k, \text{Type}_\ell, \text{Type}_\omega, \text{Type}_\omega) \mid k \in \{\mathfrak{n}, \mathfrak{e}\} \} \\
 & \cup \{ (k, \text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\ell_1 \sqcup \ell_2}) \mid k \in \{\mathfrak{n}, \mathfrak{e}\} \}
 \end{aligned}$$

■ **Figure 9** Informal rules of an Agda-like system.

The traditional encoding of inductive types using Church’s impredicative encoding looks like the following:

$$\text{NatC} = (a : \text{Prop}) \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a$$

But this is much more restrictive than the usual definition of *Nat* as a real inductive type. More specifically, when defined as an inductive type we get two extra features compared to the above Church encoding: the ability to do dependent elimination, and the ability to perform elimination to any universe rather than only to **Prop**. Let us focus on the second one. The following Church-like encoding would lift this restriction, allowing elimination to any universe:

$$\text{NatL} = (l : \text{Level}) \rightarrow (a : \text{Type}_l) \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a$$

Such a definition is possible in systems like Agda which provide the necessary universe polymorphism (the *l* above is a universe-level variable), but this type *NatL* is traditionally placed in a universe too high to be useful as an encoding of natural numbers.

We have not been able to find a concise description of the rules used in Agda, but a first approximation of its type system is described informally in Figure 9 where ω stands for the smallest infinite ordinal. According to those rules, Agda would place the above universe-polymorphic definition of *NatL* squarely in the far away Type_ω universe. Yet everything that can be done with it can also be done with the real *Nat* inductive type, which lives in the much more palatable Type_0 universe, so it would arguably be safe to let *NatL* live in Type_0 (and thus make this definition impredicative). The same reasoning applies to the following type:

$$\text{ListType} = (l : \text{Level}) \rightarrow (a : \text{Type}_l) \rightarrow a \rightarrow (\text{Type}_0 \rightarrow a \rightarrow a) \rightarrow a$$

So *ListType* should arguably live in Type_1 rather than in Type_ω since that is what happens when defined as a real inductive type. This would also make *ListType* impredicative but should not threaten consistency. This illustrates that every inductive type corresponds to an impredicative definition that could live in the same universe, making it clear that having impredicative definitions in multiple universe levels is not inherently incompatible with consistency.

Of course, this begs the question: what is it that makes it safe to let those definitions be treated as impredicative? What is special about them?

$$\begin{aligned}
\mathcal{R} = & \{ (n, l : \text{UL}, \text{Type}_\ell, \text{Type}_\omega) \} \\
& \cup \{ (e, l : \text{UL}, \text{Type}_\ell, \text{Type}_{\ell[0/l]}) \} \\
& \cup \{ (k, l : \text{UL}, \text{Type}_\omega, \text{Type}_\omega) \quad | \quad k \in \{n, e\} \} \\
& \cup \{ (k, t : \text{Type}_\ell, \text{Type}_\omega, \text{Type}_\omega) \quad | \quad k \in \{n, e\} \} \\
& \cup \{ (k, t : \text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\ell_1 \sqcup \ell_2}) \quad | \quad k \in \{n, e\} \}
\end{aligned}$$

■ **Figure 10** Informal rules of $\text{EpCC}\omega$.

In the rest of this section we will consider one hypothesis, which is that the universe level parameter ℓ needs to be erasable. In practice the vast majority of universe polymorphism can be marked as erasable. Some simple counter examples are:

$$\begin{aligned}
\text{Set} &= \lambda l : \text{Level} \rightarrow \text{Type}_l \\
\text{ListType} &= \lambda l_1 : \text{Level} \rightarrow (l_2 : \text{Level}) \rightarrow (a : \text{Type}_{l_2}) \rightarrow a \rightarrow (\text{Type}_{l_1} \rightarrow a \rightarrow a) \rightarrow a
\end{aligned}$$

4.3 $\text{EpCC}\omega$: Impredicative erasable universe polymorphism

With universe polymorphism, sorts are not closed any more, so we cannot really represent the rules that govern them using a simple set like \mathcal{R} . So, the $(k, \text{UL}, \text{Type}_\ell, \text{Type}_\omega)$ rule was really meant to say something like:

$$\frac{\Gamma \vdash \tau_1 : \text{UL} \quad \Gamma, l : \tau_1 \vdash \tau_2 : \text{Type}_\ell}{\Gamma \vdash (l : \tau_1) \xrightarrow{k} \tau_2 : \text{Type}_\omega}$$

Now if we want to make this impredicative when $k = e$, since ℓ can refer to l we need to substitute l with *something* before we can use it in the sort of the product. For the NatL case, for example, ℓ will be “s l” and we argued that this product type should live in Type_0 , so we would need to substitute l with -1 ! Rather than argue why a negative value could make sense, we will use 0 in our rule:

$$\frac{\Gamma \vdash \tau_1 : \text{UL} \quad \Gamma, l : \tau_1 \vdash \tau_2 : \text{Type}_\ell}{\Gamma \vdash (l : \tau_1) \xrightarrow{e} \tau_2 : \text{Type}_{\ell[0/l]}}$$

While this places NatL in Type_1 rather than Type_0 , it still makes it impredicative, and if all our base types live in Type_1 we will not notice much difference.

Figure 10 describes the resulting calculus we call $\text{EpCC}\omega$, where the second fields of elements of \mathcal{R} now have the shape “ $x : s$ ” so we can refer to the variable x that can appear freely in the third field.

4.4 Encoding System-F in $\text{EpCC}\omega$

$\text{EpCC}\omega$ is basically a predicative version of $\text{CC}\omega$ (hence the “p”) to which we added universe polymorphism and impredicative erasable universe polymorphism (which motivated the “E”). Contrary to the previous calculus it does not have a base impredicative universe **Prop**: its only source of impredicativity is the $(e, l : \text{UL}, \text{Type}_\ell, \text{Type}_{\ell[0/l]})$ rule which introduces the impredicative erasable universe polymorphism. Compared to Agda, it lacks inductive types but it adds a form of impredicativity. While we do not know if it is consistent, we can try and compare it to existing systems, and for that we start by showing how to encode System-F.

9:16 Is Impredicativity Implicitly Implicit?

In order for our encoding function $\llbracket \cdot \rrbracket$ to be based purely on the syntax of terms rather than the typing derivation, we take as input a stratified version of System-F:

$$\begin{aligned} (\text{types}) \quad \tau &::= t \mid \tau_1 \rightarrow \tau_2 \mid (t : *) \rightarrow \tau \\ (\text{terms}) \quad e &::= x \mid \lambda x : \tau \rightarrow e \mid e_1 e_2 \mid \lambda t : * \rightarrow e \mid e \tau \end{aligned}$$

To encode System-F, the only interesting part is the need to simulate System-F's impredicative quantification over types. We can do that in the same way $NatC$ was generalized to $NatL$, i.e. by replacing “ $(t : *) \rightarrow \tau$ ” with “ $(l : \text{Level}) \xrightarrow{e} (t : \text{Type}_l) \xrightarrow{n} \tau$ ”. The only tricky aspect of this is that while in System-F all the type variables (and more generally all the types) have the same kind $*$, this encoding makes every type variable come with its own universe level, so the encoding function needs to keep track of the level of each type in order to know how to instantiate the $(l : \text{Level}) \xrightarrow{e} \dots$ quantifiers.

The encoding function on types takes the form $\llbracket \tau \rrbracket_\Delta$ where Δ maps each type variable to its associated level variable, and it returns a pair $\tau'; \ell$ where ℓ is the universe level of τ' :

$$\begin{aligned} \llbracket t \rrbracket_\Delta &= t ; \Delta(t) \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket_\Delta &= \tau'_1 \xrightarrow{n} \tau'_2 ; \ell_1 \sqcup \ell_2 && \text{where } \tau'_1; \ell_1 = \llbracket \tau_1 \rrbracket_\Delta \text{ and } \tau'_2; \ell_2 = \llbracket \tau_2 \rrbracket_\Delta \\ \llbracket (t : *) \rightarrow \tau \rrbracket_\Delta &= (l : \text{Level}) \xrightarrow{e} (t : \text{Type}_l) \xrightarrow{n} \tau' ; \ell' && \text{where } \tau'; \ell = \llbracket \tau \rrbracket_{\Delta, t; l} \text{ and } \ell' = 1 \sqcup \ell[0/l] \end{aligned}$$

Similarly the encoding function for terms takes the form $\llbracket e \rrbracket_\Delta$:

$$\begin{aligned} \llbracket x \rrbracket_\Delta &= x \\ \llbracket \lambda x : \tau \rightarrow e \rrbracket_\Delta &= \lambda t : \tau' \xrightarrow{n} \llbracket e \rrbracket_\Delta && \text{where } \tau'; \ell = \llbracket \tau \rrbracket_\Delta \\ \llbracket e_1 e_2 \rrbracket_\Delta &= \llbracket e_1 \rrbracket_\Delta @^n \llbracket e_2 \rrbracket_\Delta \\ \llbracket \lambda t : * \rightarrow e \rrbracket_\Delta &= \lambda l : \text{Level} \xrightarrow{e} \lambda t : \text{Type}_l \xrightarrow{n} \llbracket e \rrbracket_{\Delta, t; l} \\ \llbracket e \tau \rrbracket_\Delta &= (\llbracket e \rrbracket_\Delta @^e \ell) @^n \tau' && \text{where } \tau'; \ell = \llbracket \tau \rrbracket_\Delta \end{aligned}$$

Finally we need to encode contexts as well, for which the encoding function takes the form $\llbracket \Gamma \rrbracket$ and it returns a pair $\Gamma'; \Delta$:

$$\begin{aligned} \llbracket \bullet \rrbracket &= \bullet ; \bullet \\ \llbracket \Gamma, x : \tau \rrbracket &= \Gamma', x : \llbracket \tau \rrbracket_\Delta ; \Delta && \text{where } \Gamma'; \Delta = \llbracket \Gamma \rrbracket \\ \llbracket \Gamma, t : * \rrbracket &= \Gamma', l : \text{Level}, t : \text{Type}_l ; \Delta, t : l && \text{where } \Gamma'; \Delta = \llbracket \Gamma \rrbracket \end{aligned}$$

► Theorem 7 (EpCC ω can encode System-F).

For any $\Gamma \vdash e : \tau$ in System-F, we have $\Gamma' \vdash e' : \tau'$ and $\Gamma' \vdash \tau' : \text{Type}_\ell$ in EpCC ω where $\Gamma'; \Delta = \llbracket \Gamma \rrbracket$, $e' = \llbracket e \rrbracket_\Delta$, and $\tau'; \ell = \llbracket \tau \rrbracket_\Delta$.

Proof. By structural induction on the type derivation. ◀

4.5 The power of EpCC ω

EpCC ω seems to be flexible enough to cover most uses of impredicativity found in the context of programming, such as Church's encoding, Chlipala's parametric higher-order abstract syntax [10], typed closure representations, or iCAP [25]. It does so without restricting impredicativity to a single universe, and even makes those uses more flexible in EpCC ω such as adding the equivalent of strong elimination in Church's encoding. So in this sense EpCC ω is more powerful than systems like CC ω .

Yet we have not even been able to generalize the above System-F encoding in order to encode arbitrary F_ω terms into EpCC ω . For example, consider the following F_ω term:

$$\lambda t_1 : * \rightarrow \lambda (t_2 : * \rightarrow *) \rightarrow \lambda (x : t_2 t_1) \rightarrow x$$

A simple encoding into $\text{EpCC}\omega$ could be:

$$\lambda l : \text{Level} \xrightarrow{\epsilon} \lambda t_1 : \text{Type}_{l_1} \xrightarrow{\eta} \lambda (t_2 : \text{Type}_{l_1} \xrightarrow{\eta} \text{Type}_{l_1}) \xrightarrow{\eta} \lambda x : t_2 @^n t_1 \xrightarrow{\eta} x$$

But it's not faithful to the original F_ω term because it only preserves the impredicativity of the first λ . In order to get an encoding that can work for any F_ω term, we hence need an encoding which looks like:

$$\lambda l_1 : \text{Level} \xrightarrow{\epsilon} \lambda t_1 : \text{Type}_{l_1} \xrightarrow{\eta} \lambda l_2 : \text{Level} \xrightarrow{\epsilon} \lambda t_2 : T_2 \xrightarrow{\eta} \lambda x : T_x \xrightarrow{\eta} x$$

where T_2 refers to l_2 . We can then choose T_2 and T_x as follows:

$$\begin{aligned} T_2 &= (l_3 : \text{Level}) \xrightarrow{\epsilon} \text{Type}_{l_3} \xrightarrow{\eta} \text{Type}_{l_2} \\ T_x &= t_2 @^e l_1 @^n t_1 \end{aligned}$$

This makes the term valid, but its semantics doesn't match that of the original F_ω term since we cannot pass the identity function $\lambda t : * \rightarrow t$ as f any more: its encoding would now have type $(l_3 : \text{Level}) \xrightarrow{\epsilon} \text{Type}_{l_3} \xrightarrow{\eta} \text{Type}_{l_3}$ instead of the expected $(l_3 : \text{Level}) \xrightarrow{\epsilon} \text{Type}_{l_3} \xrightarrow{\eta} \text{Type}_{l_2}$.

Similarly, we have not been able to adapt Hurkens's paradox to the $\text{EpCC}\omega$ system either. Of course, all this says is that we do not know if $\text{EpCC}\omega$ is consistent, but at least it indicates that this kind of impredicativity might be incomparable to the traditional form seen in $\text{CC}\omega$ or λU^- .

5 Related work

In [3], Augustsson presents a language where inductive types only live in the bottom universe, and shows that everything from the higher universes can be erased. This is similar to our argument in Section 3.2, but with some important differences in the universe stratification and in the definition of erasure. His universe stratification is unusual in that it is designed to keep track of erasability and does not enforce predicativity, which makes it fundamentally very different. It turns out that for $\text{eCC}\omega$ and eCIC , his stratification rules match our traditional rules when it comes to deciding if something is in the bottom universe, so his erasure should apply equally to a stratification like the one used here, although this is not the case when we consider systems like eCoq . More importantly, his notion of erasure is different from ours since his erasure of $(x : \tau_1) \xrightarrow{k} \tau_2$ is \bullet meaning that it is significantly more permissive. For example, his erasure has to be *external* (i.e., performed after checking type convertibility), whereas the erasure we use here could be *internal*, as is the case in ICC [21] and $\text{ICC}^*[5]$.

In [30], Werner discusses *internal* erasure of Coq's impredicative Prop universe. This is done in the context of the proof-irrelevance kind of erasure, where Prop is restricted to be proof-irrelevant so that it can be erased from the non- Prop universes. So this approach is contrary to ours: we erase non- Prop arguments from Prop terms, whereas he erases Prop arguments from non- Prop terms. More importantly, this kind of erasure is already present in Coq, so what Werner proposes is to make it internal, that is to take advantage of this erasure to strengthen the convertibility rule during type checking, in the same way ICC [21] and $\text{ICC}^*[5]$ systems use a stronger convertibility rule to take advantage of the kind of erasure we use here, as discussed in Section 2.2. This strengthening comes at the cost of normalization, as shown by Abel and Coquand [1].

In [15], Gilbert et.al. present a Coq and Agda library which provides a similar internal erasure of proof-irrelevant propositions. In comparison to Werner's work, they use a slightly different definition of proof-irrelevance based on *mere propositions* [27] and they get internal erasure by construction rather than by adding it to their underlying system.

In [28], Uemura shows a model of a cubical λ -calculus with a bottom universe that is impredicative and admits univalence and shows it not to satisfy the propositional resizing axiom, which applies to proof-irrelevant propositions. This puts into question the consistency of this axiom in such a calculus.

6 Conclusion

We have taken a tour of the interactions between impredicativity and erasability of arguments in EPTS. We have shown that three of the five most well known systems that admit impredicativity do it in a way that implicitly constrains all impredicative abstractions and fields to be erasable (and that the remaining two almost do it as well). We have also shown that while impredicativity and erasability seem to be correlated, erasability is neither a necessary nor a sufficient condition for impredicativity to be consistent: the inconsistency of λeU^- shows it's not sufficient, and our inability to show that UTT's impredicative definitions are all erasable suggests it's not necessary either.

It remains to be seen whether erasability as used in ECIC allows us to lift the restriction that strong elimination cannot be used on large inductive types without breaking consistency, and whether erasability as used in EpCC ω allows us to introduce a form of impredicativity applicable to all universe levels without breaking consistency.

Another avenue of research might be to try and better understand the relationship between the kind of erasure of impredicatively quantified arguments discussed here and the impredicativity of proof-irrelevant terms, as used in Coq and in the propositional resizing axiom.

References

- 1 Andreas Abel and Thierry Coquand. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality, February 2020. Submitted to Logical Methods in Computer Science. [arXiv:1911.08174](https://arxiv.org/abs/1911.08174).
- 2 Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1:29):1–36, 2012. doi:10.2168/LMCS-8(1:29)2012.
- 3 Lennart Augustsson. Cayenne – a language with dependent types. In *International Conference on Functional Programming*, page 239–250. ACM Press, September 1998. doi:10.1145/291251.289451.
- 4 Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):121–154, April 1991. doi:10.1017/S0956796800020025.
- 5 Bruno Barras and Bruno Bernardo. Implicit calculus of constructions as a programming language with dependent types. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, Budapest, Hungary, April 2008. doi:10.1007/978-3-540-78499-9_26.
- 6 Bruno Bernardo. Towards an implicit calculus of inductive constructions. extending the implicit calculus of constructions with union and subset types. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, August 2009. URL: <https://hal.inria.fr/inria-00432649>.
- 7 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22(2):1–46, 2012. doi:10.1017/S0956796812000056.
- 8 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78, August 2009. doi:10.1007/978-3-642-03359-9_6.
- 9 Luca Cardelli. Phase distinctions in type theory. DEC-SRC manuscript, 1988. URL: <https://pdfs.semanticscholar.org/4cb5/7987b78c5124bc0857155f99c11aa321546d.pdf>.

- 10 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, Victoria, BC, September 2008. doi: 10.1145/1411204.1411226.
- 11 Thierry Coquand. An analysis of Girard’s paradox. In *Annual Symposium on Logic in Computer Science*, 1986. Also published as INRIA tech-report RR-0531. URL: <https://hal.inria.fr/inria-00076023>.
- 12 Thierry Coquand. A new paradox in type theory. In *Logic, Methodology, and Philosophy of Science*, pages 7–14, 1994. doi:10.1016/S0049-237X(06)80062-5.
- 13 Thomas Fruchart and Guiseppe Longo. Carnap’s remarks on impredicative definitions and the genericity theorem. Technical Report LIENS-96-22, ENS, Paris, 1996. URL: <ftp://ftp.di.ens.fr/pub/reports/liens-96-22.A4.ps.Z>.
- 14 Herman Geuvers. (In)consistency of extensions of higher order logic and type theory. In *Types for Proofs and Programs*, pages 140–159, 2006. doi:10.1007/978-3-540-74464-1_10.
- 15 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. In *Symposium on Principles of Programming Languages*, pages 3:1–3:28. ACM Press, 2019. doi:10.1145/3290316.
- 16 Eduardo Giménez. Codifying guarded definitions with recursive schemes. Technical Report RR1995-07, École Normale Supérieure de Lyon, 1994. URL: <ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1995/RR1995-07.ps.Z>.
- 17 J. Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l’Arithmétique d’Ordre Supérieur*. PhD thesis, University of Paris VII, 1972. URL: <https://pdfs.semanticscholar.org/e1a1/c345ce8ab4c11f176f1c42bcfc6a62ef4e3c.pdf>.
- 18 Gérard P. Huet, Christine Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.
- 19 Antonius Hurkens. A simplification of Girard’s paradox. In *International conference on Typed Lambda Calculi and Applications*, pages 266–278, 1995. doi:10.1007/BFb0014058.
- 20 Zhaohui Luo. A unifying theory of dependent types: the schematic approach. In *Logical Foundations of Computer Science*, 1992. doi:10.1007/BFb0023883.
- 21 Alexandre Miquel. The implicit calculus of constructions: extending pure type systems with an intersection type binder and subtyping. In *International conference on Typed Lambda Calculi and Applications*, pages 344–359, 2001. doi:10.1007/3-540-45413-6_27.
- 22 Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364, Budapest, Hungary, April 2008. doi:10.1007/978-3-540-78499-9_25.
- 23 Stefan Monnier. The Swiss coercion. In *Programming Languages meets Program Verification*, pages 33–40, Freiburg, Germany, September 2007. ACM Press. doi:10.1145/1292597.1292604.
- 24 Stefan Monnier. Typer: ML boosted with type theory and Scheme. In *Journées Francophones des Langages Applicatifs*, pages 193–208, 2019. URL: <https://hal.inria.fr/hal-01985195/>.
- 25 Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *European Symposium on Programming*, pages 149–168, 2014. doi:10.1007/978-3-642-54833-8_9.
- 26 Matúš Tejiščák. *Erasure in Dependently Typed Programming*. PhD thesis, University of St Andrews, 2020.
- 27 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. arXiv:1308.0729.
- 28 Taichi Uemura. Cubical assemblies, a univalent and impredicative universe and a failure of propositional resizing. In *Types for Proofs and Programs*, volume 130 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:20, 2019. doi:10.4230/LIPIcs.TYPES.2018.7.
- 29 Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L’Université Paris 7, Paris, France, 1994. URL: <https://hal.inria.fr/tel-00196524/>.
- 30 Benjamin Werner. On the strength of proof-irrelevant type theories. *Logical Methods in Computer Science*, 4(3):1–20, 2008. doi:10.1007/11814771_49.

Higher Inductive Type Eliminators Without Paths

Nils Anders Danielsson 

University of Gothenburg, Sweden

Abstract

Cubical Agda has support for higher inductive types. Paths are integral to the working of this feature. However, there are other notions of equality. For instance, Cubical Agda comes with an identity type family for which the J rule computes in the usual way when applied to the canonical proof of reflexivity, whereas typical implementations of the J rule for paths do not.

This text shows how one can use some of the higher inductive types definable in Cubical Agda with arbitrary notions of equality satisfying certain axioms. The method works for several examples taken from the HoTT book, including the interval, the circle, suspensions, pushouts, the propositional truncation, a general truncation operator, and set quotients.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Cubical Agda, higher inductive types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.10

Supplementary Material Accompanying Agda code is available to download [3].

Acknowledgements I would like to thank Anders Mörtberg and Andrea Vezzosi for helping me get to grips with Cubical Agda. I would also like to thank some anonymous reviewers for useful feedback.

1 Introduction

Higher inductive types provide a way to define things like propositional truncation, (set) quotients and other things in type theory [7]. Recently support for higher inductive types has been added to Agda [1, 8]. As an example propositional truncation can be defined in the following way (where *Type a* is the universe at level *a*):

```
data ||_|| (A : Type a) : Type a where
  |_|| : A → || A ||
  trivial : (x y : || A ||) → x ≡ y
```

This type family has a regular constructor `|_||` which states that `|| A ||` is inhabited if *A* is. It also has a *higher* constructor `trivial` which states that every element of `|| A ||` is equal to every other, i.e. that `|| A ||` is a (mere) proposition. In the type of `trivial` the equality $x \equiv y$ stands for the type of *paths* from *x* to *y*. Paths in *A* are a kind of functions from the *interval* *I* to *A*. When `trivial x y` is applied to an element *i* of the interval we get a value in `|| A ||`: this value is definitionally equal to *x* if *i* is `0`, and *y* if *i* is `1`, where `0` and `1` are the endpoints of the interval. Thus all constructors of `|| A ||` target the same type.

One can define functions from `|| A ||` using pattern matching. For instance, here is a map function:

```
map : (A → B) → || A || → || B ||
map f | x |           = | f x |
map f (trivial x y i) = trivial (map f x) (map f y) i
```



© Nils Anders Danielsson;

licensed under Creative Commons License CC-BY

25th International Conference on Types for Proofs and Programs (TYPES 2019).

Editors: Marc Bezem and Assia Mahboubi; Article No. 10; pp. 10:1–10:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Higher Inductive Type Eliminators Without Paths

Let us consider the second case. Note that `trivial` is applied to three arguments. The right-hand side of the function must be an expression of type `|| B ||` that satisfies two side-conditions: if `0` is substituted for `i`, then the value must be definitionally equal to `map f x` (because `trivial x y 0` is definitionally equal to `x`), and if `1` is substituted for `i`, then the value must be definitionally equal to `map f y`. The given right-hand side satisfies these conditions.

This text is concerned with the following question: What if you want to use higher inductive types, but you do not want to use paths? Cubical Agda, the variant of Agda with support for higher inductive types, comes with two notions of equality: paths and an identity type family [2, 6]. One can prove the J rule for paths, but so far no one has managed to do this in such a way that the J rule computes in the usual way when applied to reflexivity. The identity type family comes with a J rule that does compute in the usual way when applied to reflexivity. People with code that relies on this computational behaviour of J might not want to switch to using paths. This text shows one way in which one can avoid doing this, and still make use of (at least some) higher inductive types:

- The approach works for any notion of equality that satisfies certain axioms (see Section 2). There is work in progress on adding proper support for inductive families to Cubical Agda. Such support would mean that the approach would work also for equality defined in the following way:

```
data _≡_ {A : Type a} (x : A) : A → Type a where  
  refl : x ≡ x
```

- The basic idea of the approach is to define variants of the higher constructors that use the other notion of equality instead of paths, and to define eliminators that refer to these variants of the constructors. It might seem obvious that this can be done, because any notion of equality that satisfies the axioms is equivalent to path equality. However, in Cubical Agda it is natural to express eliminators for many higher inductive types using a heterogeneous notion of path equality (see Section 4). Fortunately heterogeneous paths can be expressed using homogeneous paths (see Section 4; this result was proved together with Anders Mörtberg and Andrea Vezzosi).
- The eliminators are defined in such a way that they compute in the “right” way for constructors that do not involve paths. For higher constructors propositional “computation” rules are proved.
- The approach works for at least the following higher inductive types: the circle (see Section 5), set quotients (Section 6), the propositional truncation operator given above (Section 7), suspensions (Section 7), and some types that are not discussed in detail in the paper, but are treated in accompanying Agda code: the interval, pushouts, and a general truncation operator. The obtained eliminators are close to the induction principles given in the HoTT book [7]. No attempt is made to characterise exactly when the method works, but there is some discussion of when the method might be usable in Section 9.
- It might not come as a surprise that something like this can be done. A key contribution of the paper is, in my opinion, some functions that make it *easy*—at least for the higher inductive types mentioned above—to define the eliminators and to prove the computation rules (see Sections 4 and 5).

Dependent eliminators are defined by using the eliminators for paths, plus one of two lemmas for each higher constructor (one lemma for truncation constructors, and one for the rest). All corresponding “computation” rules (one for each higher constructor, except for the truncation constructors) are proved using the same lemma, applied to a proof of reflexivity. These three lemmas suffice for all the examples mentioned above. There are also similar lemmas for non-dependent eliminators. Section 7 demonstrates that the lemmas developed in previous sections work also for other higher inductive types.

The text is accompanied by machine-checked Agda proofs [3] (but there is no guarantee that Agda is free of bugs). Note that there are small differences between the accompanying code and the code presented below.

2 An Axiomatisation of Equality With J

Let us assume that $_ \equiv _$ has the following type:

$$_ \equiv _ : \{A : \text{Type } a\} \rightarrow A \rightarrow A \rightarrow \text{Type } a$$

(Arguments in braces are implicit arguments, that do not need to be given explicitly if Agda can infer them. To avoid clutter some implicit argument declarations, like the one for the universe level a , are omitted from type signatures.) This type family is assumed to satisfy the following axioms:

$$\begin{aligned} \text{refl} & : (x : A) \rightarrow x \equiv x \\ J & : (P : \{x y : A\} \rightarrow x \equiv y \rightarrow \text{Type } p) \rightarrow (\forall x \rightarrow P (\text{refl } x)) \rightarrow (eq : x \equiv y) \rightarrow P \text{ eq} \\ J\text{-refl} & : (P : \{x y : A\} \rightarrow x \equiv y \rightarrow \text{Type } p) (r : \forall x \rightarrow P (\text{refl } x)) \rightarrow J P r (\text{refl } x) \equiv r x \end{aligned}$$

There should be a canonical proof of reflexivity, refl , there should be a J rule, and the usual computation rule for J should hold up to the given notion of equality.

Any two notions of equality satisfying these axioms are pointwise equivalent, in the sense of the HoTT book [7], using one of the notions to define what it means to be equivalent:

$$\equiv \simeq \equiv : (x \equiv_1 y) \simeq (x \equiv_2 y)$$

(Hofmann and Streicher have proved a very similar result [4, Section 5.2].) Furthermore this proof maps refl to refl , in both directions. The proofs of these properties are easy and omitted.

Cubical Agda's path and identity type families are instances of these axioms, as is the equality type family defined as an inductive family with a single constructor refl as in Section 1. For paths this is shown in Section 3.

From now on $_ \equiv _$ will be used to refer to an arbitrary notion of equality satisfying the axioms above, whereas the path type family will be called Path . (It might be the case that $_ \equiv _$ also refers to the path type family.)

3 Homogeneous Paths

This section contains an introduction to paths, or more specifically *homogeneous* paths. Section 4 discusses *heterogeneous* paths.

The path type constructor has the following type:

$$\text{Path} : \{A : \text{Type } a\} \rightarrow A \rightarrow A \rightarrow \text{Type } a$$

The type $\text{Path } \{A = A\} x y$ (where the notation $\{A = A\}$ is used to explicitly give A as the implicit argument A) is a kind of function space from the interval I to A . It is subject to the restriction that when values in this type are applied to the endpoints of the interval, $\underline{0}$ and $\underline{1}$, we get x and y , respectively.

We can prove that the path type family is reflexive in the following way:

$$\begin{aligned} \text{refl}^P & : (x : A) \rightarrow \text{Path } x x \\ \text{refl}^P x & = \lambda _ \rightarrow x \end{aligned}$$

Note that $\text{refl}^P x i$ is equal to x for all values of i .

10:4 Higher Inductive Type Elimimators Without Paths

Cubical Agda comes with some interval operations. There is a maximum operation, max , with $\underline{0}$ as a definitional unit and $\underline{1}$ as a definitional zero. Similarly min is a minimum operation, with $\underline{1}$ as a definitional unit and $\underline{0}$ as a definitional zero. Furthermore there is a negation operation, $-$, that maps $\underline{0}$ to $\underline{1}$ and $\underline{1}$ to $\underline{0}$.

Cubical Agda also comes with a primitive transport operation:

$$transport : \{p : I \rightarrow Level\} (P : (i : I) \rightarrow Type (p i)) \rightarrow I \rightarrow P \underline{0} \rightarrow P \underline{1}$$

($Level$ is the type of universe levels.) If the interval argument is $\underline{0}$, then the computational behaviour of $transport$ depends on the type family P . However, if the interval argument is $\underline{1}$, then $transport$ returns its final argument. In this case there is a side-condition on the use of $transport$ that is not captured in its type: the type family P must be definitionally constant. (The interval argument might be an expression that does not reduce to $\underline{0}$ or $\underline{1}$, and the type family might mention interval variables used in the interval argument. In this case the application is accepted if Agda can verify that the type family is constant whenever the constraint $i = \underline{1}$ holds, where i is the interval argument [8].)

The primitive transport operation can be used to prove the J rule for paths (the notation $\{x = x\}$ is used to bind the implicit argument x to the name x):

$$\begin{aligned} J^P & : (P : \{x y : A\} \rightarrow Path x y \rightarrow Type p) \rightarrow (\forall x \rightarrow P (refl^P x)) \rightarrow \\ & (eq : Path x y) \rightarrow P eq \\ J^P \{x = x\} P p eq & = transport (\lambda i \rightarrow P (\lambda j \rightarrow eq (min i j))) \underline{0} (p x) \end{aligned}$$

Note that when i is $\underline{0}$, then the expression $P (\lambda j \rightarrow eq (min i j))$ is definitionally equal to $P (\lambda _ \rightarrow x)$, which is the type of $p x$. When i is $\underline{1}$, then the expression is definitionally equal to $P eq$.

The computation rule for J does not hold by definition for J^P . However, it can be proved using the following lemma (following Anders Mörtberg [1]):

$$\begin{aligned} transport-refl & : Path (transport (\lambda i \rightarrow refl^P A i) \underline{0}) (\lambda x \rightarrow x) \\ transport-refl \{A = A\} & = \lambda i \rightarrow transport (\lambda _ \rightarrow A) i \end{aligned}$$

Note that the first argument given to the final occurrence of $transport$ is constant when i is $\underline{1}$, as required.

Cubical Agda also has support for *composition* of paths [2, 8]. There are two variants, homogeneous and heterogeneous. Here the homogeneous variant is used to prove that path equality is transitive [2] (this can also be proved using the J rule):

$$trans^P : Path x y \rightarrow Path y z \rightarrow Path x z$$

The basic idea of the proof is to construct three sides of a square, and to use the composition operation to compute the square's fourth side. Instead of showing the Agda code I have included a diagram:

$$\begin{array}{ccc} x & \overset{trans^P \ x \equiv y \ y \equiv z \ i}{\dashrightarrow} & z \\ \uparrow x & & \uparrow y \equiv z \ j \\ x & \xrightarrow{x \equiv y \ i} & y \end{array}$$

Every arrow in the diagram is a path between the expressions at the arrow's endpoints, and the expressions between the endpoints stand for arbitrary "points" on the paths. The left-hand side of the diagram corresponds to i being $\underline{0}$, and the right-hand side to i being $\underline{1}$. Similarly, the bottom corresponds to j being $\underline{0}$, and the top to j being $\underline{1}$. The solid arrows are constructed using the two arguments given to $trans^P$ ($x \equiv y$ and $y \equiv z$), as well as a constant path (the left-hand side). The composition operation is then used to construct the dashed arrow.

The homogeneous composition operation requires that (roughly speaking) every "point" on the left and right sides of the square have the same type. The heterogeneous operation, which is used in Section 4, is more general in that it (roughly speaking) allows the types of the points on the left and right sides to vary with j .

4 Heterogeneous Paths

The path type family discussed above is a homogeneous special case of a heterogeneous notion of path:

$$Path^H : (P : I \rightarrow Type\ p) \rightarrow P\ \underline{0} \rightarrow P\ \underline{1} \rightarrow Type\ p$$

$Path$ is defined using $Path^H$:

$$\begin{aligned} Path &: \{A : Type\ a\} \rightarrow A \rightarrow A \rightarrow Type\ a \\ Path\ \{A = A\} &= Path^H\ (\lambda\ _ \rightarrow A) \end{aligned}$$

A typical eliminator for a regular inductive type takes one argument per constructor (plus some other arguments). What should the type of such an argument be for a higher constructor? It turns out that one can, at least in some cases, use heterogeneous paths to give suitable types to such arguments.

Consider the following incomplete definition of an eliminator for the propositional truncation operator:

$$\begin{aligned} elim^P &: (P : \parallel A \parallel \rightarrow Type\ p) \rightarrow ((x : A) \rightarrow P\ | x |) \rightarrow ? \rightarrow (x : \parallel A \parallel) \rightarrow P\ x \\ elim^P\ P\ f\ t\ | x | &= f\ x \\ elim^P\ P\ f\ t\ (\text{trivial}\ x\ y\ i) &= ? \end{aligned}$$

There are two side-conditions on the right-hand side of the last clause: when i is $\underline{0}$, then it must be definitionally equal to $elim^P\ P\ f\ t\ x$ (because $\text{trivial}\ x\ y\ \underline{0}$ is definitionally equal to x), and when i is $\underline{1}$, then it must be definitionally equal to $elim^P\ P\ f\ t\ y$. These requirements can be captured using $Path^H$:

$$\begin{aligned} elim^P\ P\ f\ t\ (\text{trivial}\ x\ y\ i) &= rhs\ i \\ \text{where} & \\ rhs &: Path^H\ (\lambda\ i \rightarrow P\ (\text{trivial}\ x\ y\ i))\ (elim^P\ P\ f\ t\ x)\ (elim^P\ P\ f\ t\ y) \\ rhs &= ? \end{aligned}$$

There are no side-conditions on the right-hand side of rhs . Here is a complete definition of the eliminator:

$$\begin{aligned} elim^P &: (P : \parallel A \parallel \rightarrow Type\ p) \rightarrow \\ &((x : A) \rightarrow P\ | x |) \rightarrow \\ &(\{x\ y : \parallel A \parallel\} (p : P\ x)\ (q : P\ y) \rightarrow Path^H\ (\lambda\ i \rightarrow P\ (\text{trivial}\ x\ y\ i))\ p\ q) \rightarrow \\ &(x : \parallel A \parallel) \rightarrow P\ x \end{aligned}$$

10:6 Higher Inductive Type Eliminators Without Paths

$$\begin{aligned} \text{elim}^P P f t \mid x \mid &= f x \\ \text{elim}^P P f t (\text{trivial } x y i) &= t (\text{elim}^P P f t x) (\text{elim}^P P f t y) i \end{aligned}$$

The goal here is to define eliminators that use an arbitrary notion of equality that satisfies the axioms from Section 2, not necessarily paths, either heterogeneous or homogeneous. As mentioned above homogeneous path equality is pointwise equivalent to any other notion of equality satisfying the axioms. How do heterogeneous paths fit into this picture?

Path^H can, up to equivalence, be expressed using Path :

$$\begin{aligned} \text{Path}^H &\simeq \text{Path} : \\ (P : I \rightarrow \text{Type } p) \{p : P \underline{0}\} \{q : P \underline{1}\} &\rightarrow \text{Path}^H P p q \simeq \text{Path} (\text{transport } P \underline{0} p) q \end{aligned}$$

It turns out that it is very easy to prove this equivalence.¹ One can use transport to construct the corresponding path:

$$\begin{aligned} \text{Path}^H &\equiv \text{Path} : \\ (P : I \rightarrow \text{Type } p) (p : P \underline{0}) (q : P \underline{1}) &\rightarrow \text{Path} (\text{Path}^H P p q) (\text{Path} (\text{transport } P \underline{0} p) q) \\ \text{Path}^H &\equiv \text{Path } P p q i = \\ \text{Path}^H (\lambda j \rightarrow P (\text{max } i j)) &(\text{transport } (\lambda j \rightarrow P (\text{min } i j)) (- i) p) q \end{aligned}$$

When i is $\underline{0}$, then the type family argument given to transport is constant, as required, and the right-hand side is definitionally equal to $\text{Path}^H P p q$. Furthermore, when i is $\underline{1}$, then the right-hand side is definitionally equal to $\text{Path} (\text{transport } P \underline{0} p) q$. Once the equality has been established in this way one can turn it into an equivalence by using subst^P :

$$\begin{aligned} \text{subst}^P : (P : A \rightarrow \text{Type } p) &\rightarrow \text{Path } x y \rightarrow P x \rightarrow P y \\ \text{subst}^P P x \equiv y p &= \text{transport } (\lambda i \rightarrow P (x \equiv y i)) \underline{0} p \end{aligned}$$

A function like subst^P can also be defined for the arbitrary notion of equality by using the J rule. In order to support different definitions, like subst^P for paths, let us assume that our arbitrary notion of equality comes with a function subst , along with a propositional computation rule for subst :

$$\begin{aligned} \text{subst} &: (P : A \rightarrow \text{Type } p) \rightarrow x \equiv y \rightarrow P x \rightarrow P y \\ \text{subst-refl} : \text{subst } P (\text{refl } x) p &\equiv p \end{aligned}$$

As noted in Section 2 the arbitrary notion of equality is pointwise equivalent to path equality. Let $\text{from-path} : \text{Path } x y \rightarrow x \equiv y$ denote one direction of the equivalence, and to-path the other. We can now relate subst to subst^P :

$$\text{subst} \equiv \text{subst}^P : (x \equiv y : \text{Path } x y) \rightarrow \text{subst } P (\text{from-path } x \equiv y) p \equiv \text{subst}^P P x \equiv y p$$

The proof uses the J rule for paths and the following calculation (recall that from-path maps canonical reflexivity proofs to canonical reflexivity proofs):

$$\begin{aligned} \text{subst } P (\text{from-path } (\text{refl}^P x)) p &\equiv \\ \text{subst } P (\text{refl } x) p &\equiv \\ p &\equiv \\ \text{subst}^P P (\text{refl}^P x) p &\equiv \end{aligned}$$

The last step follows from transport-refl .

¹ In retrospect. Anders Mörtberg had implemented a corresponding logical equivalence [5]. I asked Anders and Andrea Vezzosi if and how the corresponding equivalence could be proved. Andrea gave me some useful hints. I managed to finish the proof, only to find out that Andrea had proved it a couple of days before me. However, both proofs were rather complicated. The simple proof presented here was found by Anders quite some time later.

4.1 Consequences of the Equivalence

Let us now discuss some consequences of the equivalence $Path^H \simeq Path$. By combining it with $subst \equiv subst^P$ we get the following equivalence:

$$\begin{aligned} subst \equiv Path^H : \{x \equiv y : Path\ x\ y\} &\rightarrow \\ (subst\ P\ (from\text{-}path\ x \equiv y)\ p \equiv q) &\simeq Path^H\ (\lambda\ i \rightarrow P\ (x \equiv y\ i))\ p\ q \end{aligned}$$

We can calculate in the following way:

$$\begin{aligned} subst\ P\ (from\text{-}path\ x \equiv y)\ p \equiv q &\simeq \\ subst^P\ P\ x \equiv y\ p \equiv q &\simeq \\ Path\ (subst^P\ P\ x \equiv y\ p)\ q &\simeq \\ Path^H\ (\lambda\ i \rightarrow P\ (x \equiv y\ i))\ p\ q \end{aligned}$$

Thus heterogeneous paths are closely related to the dependent paths used in the types of eliminators for several higher inductive types in the HoTT book [7]. Let us denote the forward direction of the equivalence by $subst \equiv Path^H$ and the other direction by $Path^H \rightarrow subst \equiv$. Let us also give a name to the forward direction of the first two steps of the calculation above:

$$\begin{aligned} subst \equiv Path^H : \{x \equiv y : Path\ x\ y\} &\rightarrow \\ subst\ P\ (from\text{-}path\ x \equiv y)\ p \equiv q &\rightarrow Path\ (subst^P\ P\ x \equiv y\ p)\ q \end{aligned}$$

The HoTT book also makes use of a function called *apd*, defined using *J* [7]. Let us assume that the arbitrary notion of equality comes with such a function, along with a propositional computation rule:

$$\begin{aligned} cong^D : (f : (x : A) \rightarrow P\ x)\ (x \equiv y : x \equiv y) &\rightarrow subst\ P\ x \equiv y\ (f\ x) \equiv f\ y \\ cong^D\text{-}refl : (f : (x : A) \rightarrow P\ x) &\rightarrow cong^D\ f\ (refl\ x) \equiv subst\text{-}refl \end{aligned}$$

We can prove a similar property for paths [5]:

$$\begin{aligned} cong^H : (f : (x : A) \rightarrow P\ x)\ (x \equiv y : Path\ x\ y) &\rightarrow Path^H\ (\lambda\ i \rightarrow P\ (x \equiv y\ i))\ (f\ x)\ (f\ y) \\ cong^H\ f\ x \equiv y\ i &= f\ (x \equiv y\ i) \end{aligned}$$

The functions can be related in the following way:

$$\begin{aligned} cong^D \equiv cong^H : \\ \{x \equiv y : Path\ x\ y\}\ (f : (x : A) \rightarrow P\ x) &\rightarrow \\ cong^D\ f\ (from\text{-}path\ x \equiv y) &\equiv Path^H \rightarrow subst \equiv\ (cong^H\ f\ x \equiv y) \end{aligned}$$

We can prove $cong^D \equiv cong^H$ by defining $cong^{DP}$ (a variant of $cong^D$ for paths) and relating this variant to $cong^D$ as well as $cong^H$.

Given the definition of $subst^P$ above one can define $cong^{DP}$ using *transport* in the following way:

$$\begin{aligned} cong^{DP} : (f : (x : A) \rightarrow P\ x)\ (x \equiv y : Path\ x\ y) &\rightarrow Path\ (subst^P\ P\ x \equiv y\ (f\ x))\ (f\ y) \\ cong^{DP}\ \{P = P\}\ f\ x \equiv y &= \lambda\ i \rightarrow transport\ (\lambda\ j \rightarrow P\ (x \equiv y\ (max\ i\ j)))\ i\ (f\ (x \equiv y\ i)) \end{aligned}$$

The proof of the following property relating $cong^D$ and $cong^{DP}$ is omitted (see the accompanying code for details):

$$\begin{aligned} cong^D \equiv cong^{DP} : \\ \{x \equiv y : Path\ x\ y\} &\rightarrow \\ subst \equiv Path^H \equiv (cong^D\ f\ (from\text{-}path\ x \equiv y)) &\equiv cong^{DP}\ f\ x \equiv y \end{aligned}$$

10:8 Higher Inductive Type Eliminators Without Paths

Let us instead focus on the proof of the following property that relates cong^{DP} to cong^H ($_ \simeq _ .to$ gives the forward direction of an equivalence, and $_ \simeq _ .from$ the other one):

$$\begin{aligned} & \text{cong}^{DP} \equiv \text{cong}^H : \\ & \{x \equiv y : \text{Path } x \ y\} (f : (x : A) \rightarrow P \ x) \rightarrow \\ & \text{Path } (\text{cong}^{DP} f \ x \equiv y) (_ \simeq _ .to (\text{Path}^H \simeq \text{Path } (\lambda \ i \rightarrow P \ (x \equiv y \ i))) (\text{cong}^H f \ x \equiv y)) \end{aligned}$$

We can start by using the J rule for the path $x \equiv y$, and then calculate in the following way:

$$\begin{aligned} & \text{cong}^{DP} f (\text{refl}^P x) && \equiv \\ & (\lambda \ i \rightarrow \text{transport } (\lambda _ \rightarrow P \ x) \ i \ (f \ x)) && \equiv \\ & \text{transport } (\lambda \ i \rightarrow \text{Path } (\text{transport } (\lambda _ \rightarrow P \ x) \ (- \ i) \ (f \ x)) \ (f \ x)) \ \underline{0} \ (\text{refl}^P (f \ x)) && \equiv \\ & \text{transport } (\lambda \ i \rightarrow \text{Path } (\text{transport } (\lambda _ \rightarrow P \ x) \ (- \ i) \ (f \ x)) \ (f \ x)) \ \underline{0} \\ & \quad (\text{transport } (\lambda _ \rightarrow \text{Path } (f \ x) \ (f \ x)) \ \underline{0} \ (\text{refl}^P (f \ x))) && \equiv \\ & _ \simeq _ .to (\text{Path}^H \simeq \text{Path } (\lambda \ i \rightarrow P \ (\text{refl}^P x \ i))) (\text{cong}^H f (\text{refl}^P x)) \end{aligned}$$

The first and last steps hold by definition. The third step follows from *transport-refl*. Finally the second step uses heterogeneous composition to construct the dashed arrow of the following square:

$$\begin{array}{ccc} & \text{transport } (\lambda \ i \rightarrow \text{Path } (\text{transport } (\lambda _ \rightarrow P \ x) \ (- \ i) \ (f \ x)) \ (f \ x)) \ \underline{0} \ (\text{refl}^P (f \ x)) & \\ & \uparrow \text{dashed} & \uparrow r \\ \lambda \ i \rightarrow \text{transport } (\lambda _ \rightarrow P \ x) \ i \ (f \ x) & & \\ & \uparrow l & \\ \text{refl}^P (f \ x) & \xrightarrow{b} & \text{transport } (\lambda _ \rightarrow \text{Path } (f \ x) \ (f \ x)) \ \underline{0} \ (\text{refl}^P (f \ x)) \end{array}$$

The bottom line in the diagram (*b*) is defined in the following way:

$$\text{transport } (\lambda _ \rightarrow \text{Path } (f \ x) \ (f \ x)) \ (- \ i) \ (\text{refl}^P (f \ x))$$

The left-hand side of the diagram (*l*) corresponds to i being $\underline{0}$, and the right-hand side (*r*) to i being $\underline{1}$. Similarly, the bottom (*b*) corresponds to j being $\underline{0}$, and the dashed arrow to j being $\underline{1}$. The diagram's left-hand side (*l*) is defined in the following way:

$$\lambda \ k \rightarrow \text{transport } (\lambda _ \rightarrow P \ x) \ (\text{max } k \ (- \ j)) \ (f \ x)$$

The right-hand side (*r*) is defined in the following way:

$$\text{transport } (\lambda \ k \rightarrow \text{Path } (\text{transport } (\lambda _ \rightarrow P \ x) \ (- \ \text{min } k \ j) \ (f \ x)) \ (f \ x)) \ \underline{0} \ (\text{refl}^P (f \ x))$$

As mentioned above the heterogeneous composition operation allows the types of *l* and *r* to vary with j . In this case these expressions have the following type:

$$\text{Path } (\text{transport } (\lambda _ \rightarrow P \ x) \ (- \ j) \ (f \ x)) \ (f \ x)$$

With $\text{cong}^D \equiv \text{cong}^{DP}$ and $\text{cong}^{DP} \equiv \text{cong}^H$ in place it is easy to prove $\text{cong}^D \equiv \text{cong}^H$. As a corollary we get the following property that will be used below:

$$\begin{aligned} & \text{dependent-computation-rule-lemma} : \\ & \{x \equiv y : \text{Path } x \ y\} \{fx \equiv fy : \text{subst } P \ (\text{from-path } x \equiv y) \ (f \ x) \equiv f \ y\} \rightarrow \\ & \text{cong}^H f \ x \equiv y \equiv \text{subst} \Rightarrow \text{Path}^H \ fx \equiv fy \rightarrow \text{cong}^D f \ (\text{from-path } x \equiv y) \equiv fx \equiv fy \end{aligned}$$

5 The Circle Without Paths

Let us now see how we can make the definition of a higher inductive type—the circle [7]—usable with the arbitrary notion of equality satisfying the axioms from Section 2 (and with *subst*, *subst-refl*, *cong^D* and *cong^D-refl* instantiated in some way, as discussed in Section 4).

Here is the definition of the circle, using paths:

```
data  $\mathbb{S}^1$  : Type where
  base :  $\mathbb{S}^1$ 
  loopP : Path base base
```

It is easy to define a variant of *loop^P* that uses the arbitrary notion of equality instead of a path:

```
loop : base  $\equiv$  base
loop = from-path loopP
```

What about the eliminator? An eliminator that uses paths can be defined in the following way:

```
elimP : (P :  $\mathbb{S}^1 \rightarrow$  Type p) (b : P base)  $\rightarrow$  PathH ( $\lambda$  i  $\rightarrow$  P (loopP i)) b b  $\rightarrow$  (x :  $\mathbb{S}^1$ )  $\rightarrow$  P x
elimP P b  $\ell$  base = b
elimP P b  $\ell$  (loopP i) =  $\ell$  i
```

Now it is easy to use *subst \equiv Path^H* to construct an eliminator that uses the arbitrary notion of equality instead. The type signature matches the one given in the HoTT book [7]:

```
elim : (P :  $\mathbb{S}^1 \rightarrow$  Type p) (b : P base) ( $\ell$  : subst P loop b  $\equiv$  b) (x :  $\mathbb{S}^1$ )  $\rightarrow$  P x
elim P b  $\ell$  = elimP P b (subst $\equiv$ PathH  $\ell$ )
```

The HoTT book gives two computation rules for the eliminator. The one for the point constructor **base** is stated to be definitional, and that is the case here. The one for the higher constructor is given as an equality, and we can do the same thing:

```
elim-loop : congD (elim P b  $\ell$ ) loop  $\equiv$   $\ell$ 
elim-loop = dependent-computation-rule-lemma (refl  $\_$ )
```

The proof simply applies *dependent-computation-rule-lemma* to reflexivity. Things have been set up in such a way that *cong^H* (*elim* *P* *b* ℓ) *loop^P* is definitionally equal to *subst \equiv Path^H* ℓ ; every step of the following calculation holds by definition:

```
congH (elim P b  $\ell$ ) loopP  $\equiv$ 
( $\lambda$  i  $\rightarrow$  elim P b  $\ell$  (loopP i))  $\equiv$ 
( $\lambda$  i  $\rightarrow$  elimP P b (subst $\equiv$ PathH  $\ell$ ) (loopP i))  $\equiv$ 
( $\lambda$  i  $\rightarrow$  subst $\equiv$ PathH  $\ell$  i)  $\equiv$ 
subst $\equiv$ PathH  $\ell$ 
```

We can also define a non-dependent eliminator. This definition does not require most of the machinery introduced above. Here is a non-dependent eliminator for paths:

```
recP : (b : A)  $\rightarrow$  Path b b  $\rightarrow$   $\mathbb{S}^1 \rightarrow$  A
recP = elimP  $\_$ 
```

10:10 Higher Inductive Type Eliminators Without Paths

This variant can be used to define an eliminator for the arbitrary notion of equality:

$$\begin{aligned} \text{rec} & : (b : A) \rightarrow b \equiv b \rightarrow \mathbb{S}^1 \rightarrow A \\ \text{rec } b \ell & = \text{rec}^P b (\text{to-path } \ell) \end{aligned}$$

We simply convert the equality to a path. The computation rule for the higher constructor is stated using *cong*, a function that, along with a propositional computation rule, is assumed to come with our arbitrary notion of equality (these functions could be defined using J):

$$\begin{aligned} \text{cong} & : (f : A \rightarrow B) \rightarrow x \equiv y \rightarrow f x \equiv f y \\ \text{cong-refl} & : \text{cong } f (\text{refl } x) \equiv \text{refl } (f x) \end{aligned}$$

The computation rule can be stated and proved in the following way:

$$\begin{aligned} \text{rec-loop} & : \text{cong } (\text{rec } b \ell) \text{ loop} \equiv \ell \\ \text{rec-loop} & = \text{non-dependent-computation-rule-lemma } (\text{refl } _) \end{aligned}$$

Here *non-dependent-computation-rule-lemma* is a lemma that is easy to prove:

$$\begin{aligned} \text{non-dependent-computation-rule-lemma} & : \\ & \{x \equiv y : \text{Path } x y\} \{fx \equiv fy : f x \equiv f y\} \rightarrow \\ & \text{cong}^H f x \equiv y \equiv \text{to-path } fx \equiv fy \rightarrow \text{cong } f (\text{from-path } x \equiv y) \equiv fx \equiv fy \end{aligned}$$

An alternative is to define the non-dependent eliminator in terms of the dependent one:

$$\begin{aligned} \text{rec}' & : (b : A) \rightarrow b \equiv b \rightarrow \mathbb{S}^1 \rightarrow A \\ \text{rec}' b \ell & = \text{elim } _ b (\text{trans } \text{subst-const } \ell) \end{aligned}$$

Here *trans* and *subst-const* have the following types:

$$\begin{aligned} \text{trans} & : x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z \\ \text{subst-const} & : \text{subst } (\lambda _ \rightarrow A) x \equiv y z \equiv z \end{aligned}$$

The third argument to *elim* thus captures the following calculation, where the first step uses *subst-const* and the second uses *ℓ*:

$$\begin{aligned} \text{subst } (\lambda _ \rightarrow A) \text{ loop } b & \equiv \\ b & \equiv \\ b & \end{aligned}$$

The computation rule can be proved using the computation rule for the dependent eliminator:

$$\begin{aligned} \text{rec}'\text{-loop} & : \text{cong } (\text{rec}' b \ell) \text{ loop} \equiv \ell \\ \text{rec}'\text{-loop} & = \text{cong}^{D \equiv \rightarrow} \text{cong} \equiv \text{elim-loop} \end{aligned}$$

The proof uses the following lemma:

$$\begin{aligned} \text{cong}^{D \equiv \rightarrow} \text{cong} \equiv & : \\ & \{x \equiv y : x \equiv y\} \{fx \equiv fy : f x \equiv f y\} \rightarrow \\ & \text{cong}^D f x \equiv y \equiv \text{trans } \text{subst-const } fx \equiv fy \rightarrow \text{cong } f x \equiv y \equiv fx \equiv fy \end{aligned}$$

6 Set Quotients Without Paths

The higher inductive type given for the circle does not include any truncation constructor, i.e. a constructor that states directly that the type has a certain *h-level*. As an example of such a higher inductive type this section treats set quotients [7], which come with a truncation constructor that ensures that the resulting types are sets (in the sense of the HoTT book [7]).

A type of h-level n is an $(n - 2)$ -type:

```

Contractible : Type a → Type a
Contractible A = Σ A λ x → (y : A) → x ≡ y

H-level : ℕ → Type a → Type a
H-level zero      A = Contractible A
H-level (suc zero) A = (x y : A) → x ≡ y
H-level (suc (suc n)) A = {x y : A} → H-level (suc n) (x ≡ y)

```

Propositions (or mere propositions) are types of h-level 1, and sets are types of h-level 2:

```

Is-proposition : Type a → Type a
Is-proposition = H-level 1

Is-set : Type a → Type a
Is-set = H-level 2

```

Let $H\text{-level}^P$, $Is\text{-proposition}^P$ and $Is\text{-set}^P$ refer to the corresponding concepts defined using paths instead of the arbitrary notion of equality.

Now we can define set quotients (the definition is similar to the one in the HoTT book [7], but the relations are not required to be propositional, following Mörtberg [5]):

```

data _/_ (A : Type a) (R : A → A → Type r) : Type (a ⊔ r) where
  []      : A → A / R
  []-respects-relationP : R x y → Path [ x ] [ y ]
  /-is-setP      : Is-setP (A / R)

```

(Here $_ \sqcup _$ is a maximum operator for universe levels.) The constructor $[_]$ —`box`—takes values from the underlying type to the quotient, and the type of $[_]$ -`respects-relation`^P implies that `box` maps related values to equal values. If we expand $Is\text{-set}^P$, then we see that the $_/-is-set^P$ constructor takes two paths between quotient values, and returns a path between paths:

$$\{x\ y : A / R\} (eq_1\ eq_2 : Path\ x\ y) \rightarrow Path\ eq_1\ eq_2$$

A direct definition of an eliminator could take the following form:

```

elimP' : (P : A / R → Type p)
  (f : ∀ x → P [ x ])
  (g : ∀ {x y} (r : R x y) → PathH (λ i → P ([]-respects-relationP r i)) (f x) (f y)) →
  (∀ {x y} {eq1 eq2 : Path x y} {p : P x} {q : P y}
    (eq3 : PathH (λ i → P (eq1 i)) p q) (eq4 : PathH (λ i → P (eq2 i)) p q) →
    PathH (λ i → PathH (λ j → P (/is-setP eq1 eq2 i j)) p q) eq3 eq4) →
  (x : A / R) → P x

```

10:12 Higher Inductive Type Eliminators Without Paths

$$\begin{aligned}
\mathit{elim}^{P'} P f g h [x] &= f x \\
\mathit{elim}^{P'} P f g h ([\text{-respects-relation}^P r i]) &= g r i \\
\mathit{elim}^{P'} P f g h (/ \text{-is-set}^P p q i j) &= \\
h (\lambda i \rightarrow \mathit{elim}^{P'} P f g h (p i)) (\lambda i \rightarrow \mathit{elim}^{P'} P f g h (q i)) i j
\end{aligned}$$

However, the type of the penultimate argument might look somewhat daunting. It can be replaced by the requirement that the motive P is a family of sets. (For performance reasons the accompanying code defines the eliminator in a slightly different way: the second, third and fourth arguments are bundled up using a record type with η -equality turned off. Other eliminators from this section are also defined in this way in the accompanying code.)

Before defining an alternative eliminator, let us prove some lemmas. Consider the following variant of the final clause of $H\text{-level}^P$:

$$\begin{aligned}
&H\text{-level}^P\text{-suc} \simeq H\text{-level}^P\text{-Path}^H : \\
&\{P : I \rightarrow \text{Type } p\} \rightarrow \\
&H\text{-level}^P (\text{suc } n) (P i) \simeq ((x : P \underline{0}) (y : P \underline{1}) \rightarrow H\text{-level}^P n (Path^H P x y))
\end{aligned}$$

This variant can be proved by calculating in the following way:

$$\begin{aligned}
H\text{-level}^P (\text{suc } n) (P i) &\simeq \\
H\text{-level}^P (\text{suc } n) (P \underline{1}) &\simeq \\
((x y : P \underline{1}) \rightarrow H\text{-level}^P n (x \equiv y)) &\simeq \\
((x : P \underline{0}) (y : P \underline{1}) \rightarrow H\text{-level}^P n (Path (transport P \underline{0} x) y)) &\simeq \\
((x : P \underline{0}) (y : P \underline{1}) \rightarrow H\text{-level}^P n (Path^H P x y))
\end{aligned}$$

The first step follows from the following equality:

$$\begin{aligned}
&\mathit{index\text{-irrelevant}} : (P : I \rightarrow \text{Type } p) \rightarrow \forall i j \rightarrow Path (P i) (P j) \\
&\mathit{index\text{-irrelevant}} P i j k = P (\max (\min i (- k)) (\min j k))
\end{aligned}$$

The second step is related to Lemma 3.11.10 from the HoTT book [7], and the fourth step uses $Path^H \simeq Path$. The third step uses $\mathit{index\text{-irrelevant}}$ again, as well as the following preservation lemma:

$$\begin{aligned}
&\Pi\text{-cong} : (A \simeq B : A \simeq B) \rightarrow (\forall x \rightarrow P x \simeq Q (_ \simeq _ \text{.to } A \simeq B x)) \rightarrow \\
&((x : A) \rightarrow P x) \simeq ((x : B) \rightarrow Q x)
\end{aligned}$$

This step also uses $\mathit{transport\text{-refl}}$ and the following equality:

$$\begin{aligned}
&\mathit{transport\text{-transport}} : (P : I \rightarrow \text{Type } p) \{p : P \underline{0}\} \rightarrow \\
&Path (transport (\lambda i \rightarrow P (- i)) \underline{0} (transport P \underline{0} p)) p
\end{aligned}$$

We can use $H\text{-level}^P\text{-suc} \simeq H\text{-level}^P\text{-Path}^H$ to prove that a heterogeneous notion of proof irrelevance holds for families of propositions:

$$\begin{aligned}
&\mathit{heterogeneous\text{-irrelevance}} : \\
&(\forall x \rightarrow \mathit{Is\text{-proposition}}^P (P x)) \rightarrow \\
&\{x \equiv y : Path x y\} \{p : P x\} \{q : P y\} \rightarrow Path^H (\lambda i \rightarrow P (x \equiv y i)) p q
\end{aligned}$$

We can reason in the following way:

$$\begin{aligned}
(\forall x \rightarrow \mathit{Is\text{-proposition}}^P (P x)) &\rightarrow \\
\mathit{Is\text{-proposition}}^P (P x) &\rightarrow
\end{aligned}$$

$$\begin{aligned}
& \text{Is-proposition}^P (P (x \equiv y \underline{0})) && \rightarrow \\
& \text{Contractible}^P (\text{Path}^H (\lambda i \rightarrow P (x \equiv y i)) p q) && \rightarrow \\
& \text{Path}^H (\lambda i \rightarrow P (x \equiv y i)) p q
\end{aligned}$$

The third step follows from $H\text{-level}^P\text{-suc} \simeq H\text{-level}^P\text{-Path}^H$, and the last step uses the fact that contractible types are inhabited. We can also prove a similar result for families of sets:²

$$\begin{aligned}
& \text{heterogeneous-UIP} : \\
& (\forall x \rightarrow \text{Is-set}^P (P x)) \rightarrow \\
& \{eq_1 eq_2 : \text{Path } x y\} \{eq_3 : \text{Path } eq_1 eq_2\} \{p_1 : P x\} \{p_2 : P y\} \\
& (eq_4 : \text{Path}^H (\lambda j \rightarrow P (eq_1 j)) p_1 p_2) \\
& (eq_5 : \text{Path}^H (\lambda j \rightarrow P (eq_2 j)) p_1 p_2) \rightarrow \\
& \text{Path}^H (\lambda i \rightarrow \text{Path}^H (\lambda j \rightarrow P (eq_3 i j)) p_1 p_2) eq_4 eq_5
\end{aligned}$$

The proof is very similar to the previous one:

$$\begin{aligned}
& (\forall x \rightarrow \text{Is-set}^P (P x)) && \rightarrow \\
& \text{Is-set}^P (P x) && \rightarrow \\
& \text{Is-set}^P (P (eq_3 \underline{0} \underline{0})) && \rightarrow \\
& \text{Is-proposition}^P (\text{Path}^H (\lambda j \rightarrow P (eq_3 \underline{0} j)) p_1 p_2) && \rightarrow \\
& \text{Contractible}^P (\text{Path}^H (\lambda i \rightarrow \text{Path}^H (\lambda j \rightarrow P (eq_3 i j)) p_1 p_2) eq_4 eq_5) && \rightarrow \\
& \text{Path}^H (\lambda i \rightarrow \text{Path}^H (\lambda j \rightarrow P (eq_3 i j)) p_1 p_2) eq_4 eq_5
\end{aligned}$$

Here $H\text{-level}^P\text{-suc} \simeq H\text{-level}^P\text{-Path}^H$ is used twice, once in the third step and once in the fourth.

Now let us go back to the set quotients. Using *heterogeneous-UIP* it is easy to implement the following dependent eliminator and a corresponding non-dependent eliminator:

$$\begin{aligned}
& \text{elim}^P : (P : A / R \rightarrow \text{Type } p) \\
& (f : \forall x \rightarrow P [x]) \rightarrow \\
& (\forall \{x y\} (r : R x y) \rightarrow \text{Path}^H (\lambda i \rightarrow P ([]\text{-respects-relation}^P r i)) (f x) (f y)) \rightarrow \\
& (\forall x \rightarrow \text{Is-set}^P (P x)) \rightarrow \\
& (x : A / R) \rightarrow P x \\
& \text{elim}^P P f g s = \text{elim}^{P'} P f g (\text{heterogeneous-UIP } s) \\
& \text{rec}^P : (f : A \rightarrow B) \rightarrow (\forall \{x y\} \rightarrow R x y \rightarrow \text{Path } (f x) (f y)) \rightarrow \text{Is-set}^P B \rightarrow A / R \rightarrow B \\
& \text{rec}^P f g s = \text{elim}^P _ f g (\lambda _ \rightarrow s)
\end{aligned}$$

With the non-dependent eliminator one can define a function from A / R to a set B by giving a function from A to B that respects the relation.

Let us now define variants without paths of the higher constructors and the last two eliminators. The first higher constructor can be treated in the same way as before:

$$\begin{aligned}
& []\text{-respects-relation} : R x y \rightarrow _ \equiv _ \{A = A / R\} [x] [y] \\
& []\text{-respects-relation} = \text{from-path} \circ []\text{-respects-relation}^P
\end{aligned}$$

For the other one we can start by noting that the two definitions of h-levels given above are pointwise equivalent:

$$H\text{-level} \simeq H\text{-level}^P : \forall n \rightarrow H\text{-level } n A \simeq H\text{-level}^P n A$$

² Zesen Qian has proved more or less the same result, but in a different way [5, Git commit 9a4f3cf3c733db82344bfc98b82f405101df816a].

10:14 Higher Inductive Type Eliminators Without Paths

It is then easy to define the variant of the second constructor:

$$\begin{aligned} /-is-set &: Is-set (A / R) \\ /-is-set &= _ \simeq _ .from (H-level \simeq H-level^P 2) /-is-set^P \end{aligned}$$

The dependent eliminator can be defined in the following way, using $subst \equiv \rightarrow Path^H$ and $H-level \simeq H-level^P$:

$$\begin{aligned} elim &: (P : A / R \rightarrow Type p) \\ & (f : \forall x \rightarrow P [x]) \rightarrow \\ & (\forall \{x y\} (r : R x y) \rightarrow subst P ([]-respects-relation r) (f x) \equiv f y) \rightarrow \\ & (\forall x \rightarrow Is-set (P x)) \rightarrow \\ & (x : A / R) \rightarrow P x \\ elim P f g s &= elim^P P f (subst \equiv \rightarrow Path^H \circ g) (_ \simeq _ .to (H-level \simeq H-level^P 2) \circ s) \end{aligned}$$

Finally it is easy to define a variant of rec^P :

$$\begin{aligned} rec &: (f : A \rightarrow B) \rightarrow (\forall \{x y\} \rightarrow R x y \rightarrow f x \equiv f y) \rightarrow Is-set B \rightarrow A / R \rightarrow B \\ rec f g s &= rec^P f (to-path \circ g) (_ \simeq _ .to (H-level \simeq H-level^P 2) s) \end{aligned}$$

I have not included computation rules for the higher constructors, because sets have propositional equality types (i.e. any two equality proofs of the same type are equal).

7 More Examples

Let us now consider more examples. No new functionality is introduced in this section: the functions introduced above suffice to handle a large number of examples.

The suspension type constructor and corresponding eliminators can be defined in the following way [7]:

$$\begin{aligned} \mathbf{data} \text{ Susp } (A : Type a) &: Type a \mathbf{where} \\ \text{north} &: \text{Susp } A \\ \text{south} &: \text{Susp } A \\ \text{meridian}^P &: A \rightarrow \text{Path north south} \\ \\ elim^P &: (P : \text{Susp } A \rightarrow Type p) (n : P \text{ north}) (s : P \text{ south}) \rightarrow \\ & (\forall x \rightarrow \text{Path}^H (\lambda i \rightarrow P (\text{meridian}^P x i)) n s) \rightarrow \\ & (x : \text{Susp } A) \rightarrow P x \\ elim^P _ n s n \equiv s \text{ north} &= n \\ elim^P _ n s n \equiv s \text{ south} &= s \\ elim^P _ n s n \equiv s (\text{meridian}^P x i) &= n \equiv s x i \\ \\ rec^P &: (n s : B) \rightarrow (A \rightarrow \text{Path } n s) \rightarrow \text{Susp } A \rightarrow B \\ rec^P &= elim^P _ \end{aligned}$$

Variants of the higher constructor and the eliminators, and two propositional computation rules, can then be defined in the following way:

$$\begin{aligned} \text{meridian} &: A \rightarrow _ \equiv _ \{A = \text{Susp } A\} \text{ north south} \\ \text{meridian} &= \text{from-path} \circ \text{meridian}^P \end{aligned}$$

$$\begin{aligned}
& \text{elim} : (P : \text{Susp } A \rightarrow \text{Type } p) (n : P \text{ north}) (s : P \text{ south}) \rightarrow \\
& \quad (\forall x \rightarrow \text{subst } P (\text{meridian } x) n \equiv s) \rightarrow \\
& \quad (x : \text{Susp } A) \rightarrow P x \\
& \text{elim } P n s n \equiv s = \text{elim}^P P n s (\text{subst} \equiv \rightarrow \text{Path}^H \circ n \equiv s) \\
& \text{elim-meridian} : (P : \text{Susp } A \rightarrow \text{Type } p) (n : P \text{ north}) (s : P \text{ south}) \\
& \quad (n \equiv s : \forall x \rightarrow \text{subst } P (\text{meridian } x) n \equiv s) \rightarrow \\
& \quad \text{cong}^D (\text{elim } P n s n \equiv s) (\text{meridian } x) \equiv n \equiv s x \\
& \text{elim-meridian } _ _ _ = \text{dependent-computation-rule-lemma} (\text{refl } _) \\
& \text{rec} : (n s : B) \rightarrow (A \rightarrow n \equiv s) \rightarrow \text{Susp } A \rightarrow B \\
& \text{rec } n s n \equiv s = \text{rec}^P n s (\text{to-path} \circ n \equiv s) \\
& \text{rec-meridian} : (n s : B) (n \equiv s : A \rightarrow n \equiv s) \rightarrow \\
& \quad \text{cong} (\text{rec } n s n \equiv s) (\text{meridian } x) \equiv n \equiv s x \\
& \text{rec-meridian } _ _ _ = \text{non-dependent-computation-rule-lemma} (\text{refl } _)
\end{aligned}$$

Note that most of the text consists of type signatures, and that the lemmas introduced above can be used unchanged.

As a second example of a higher inductive type with a truncation constructor, let us now return to the propositional truncation operator from Section 1 (with `trivial` renamed to `trivialP`):

```

data ||_|| (A : Type a) : Type a where
  |_|      : A → || A ||
  trivialP : (x y : || A ||) → Path x y

```

The following eliminator was given in Section 4 (but it was called `elimP`):

$$\begin{aligned}
& \text{elim}^{P'} : (P : || A || \rightarrow \text{Type } p) \rightarrow \\
& \quad ((x : A) \rightarrow P | x |) \rightarrow \\
& \quad (\{x y : || A ||\} (p : P x) (q : P y) \rightarrow \text{Path}^H (\lambda i \rightarrow P (\text{trivial}^P x y i)) p q) \rightarrow \\
& \quad (x : || A ||) \rightarrow P x \\
& \text{elim}^{P'} P f t | x | = f x \\
& \text{elim}^{P'} P f t (\text{trivial}^P x y i) = t (\text{elim}^{P'} P f t x) (\text{elim}^{P'} P f t y) i
\end{aligned}$$

Just like for the set quotients in Section 6 we can define an alternative eliminator and a corresponding non-dependent eliminator:

$$\begin{aligned}
& \text{elim}^P : (P : || A || \rightarrow \text{Type } p) \rightarrow \\
& \quad ((x : A) \rightarrow P | x |) \rightarrow \\
& \quad (\forall x \rightarrow \text{Is-proposition}^P (P x)) \rightarrow \\
& \quad (x : || A ||) \rightarrow P x \\
& \text{elim}^P P f p = \text{elim}^{P'} P f (\lambda _ _ \rightarrow \text{heterogeneous-irrelevance } p) \\
& \text{rec}^P : (A \rightarrow B) \rightarrow \text{Is-proposition}^P B \rightarrow || A || \rightarrow B \\
& \text{rec}^P f p = \text{elim}^P _ f (\lambda _ \rightarrow p)
\end{aligned}$$

Variants of the higher constructor and the eliminators can then be defined in the following way:

```

trivial : Is-proposition || A ||
trivial = _≃_.from (H-level ≃ H-levelP 1) trivialP

```

10:16 Higher Inductive Type Eliminators Without Paths

$$\begin{aligned}
& \text{elim} : (P : \parallel A \parallel \rightarrow \text{Type } p) \rightarrow \\
& \quad ((x : A) \rightarrow P \mid x \mid) \rightarrow \\
& \quad (\forall x \rightarrow \text{Is-proposition } (P \ x)) \rightarrow \\
& \quad (x : \parallel A \parallel) \rightarrow P \ x \\
& \text{elim } P \ f \ p = \text{elim}^P \ P \ f \ (_ \simeq _ . \text{to } (H\text{-level} \simeq H\text{-level}^P \ 1) \circ p) \\
& \\
& \text{rec} : (A \rightarrow B) \rightarrow \text{Is-proposition } B \rightarrow \parallel A \parallel \rightarrow B \\
& \text{rec } f \ p = \text{rec}^P \ f \ (_ \simeq _ . \text{to } (H\text{-level} \simeq H\text{-level}^P \ 1) \ p)
\end{aligned}$$

Note again that the lemmas introduced above can be used unchanged. (Computation rules for the higher constructors are omitted, because propositions have propositional equality types.)

8 An Alternative Approach

This section discusses an alternative approach, suggested by an anonymous reviewer. The circle is used as an example.

We can write down the formation, introduction, elimination and computation rules of the circle using Σ -types in the following way:

$$\begin{aligned}
& \text{Circle} : (p : \text{Level}) \rightarrow \text{Type } (\text{lsuc } p) \\
& \text{Circle } p = \\
& \quad \Sigma \text{Type} \quad \lambda \mathbb{S}^1 \rightarrow \\
& \quad \Sigma \mathbb{S}^1 \quad \lambda \text{base} \rightarrow \\
& \quad \Sigma (\text{base} \equiv \text{base}) \lambda \text{loop} \rightarrow \\
& \quad (P : \mathbb{S}^1 \rightarrow \text{Type } p) (b : P \ \text{base}) (\ell : \text{subst } P \ \text{loop } b \equiv b) \rightarrow \\
& \quad \Sigma ((x : \mathbb{S}^1) \rightarrow P \ x) \lambda \text{elim} \rightarrow \\
& \quad \Sigma (\text{elim } \text{base} \equiv b) \lambda \text{elim-base} \rightarrow \\
& \quad \text{subst } (\lambda b \rightarrow \text{subst } P \ \text{loop } b \equiv b) \ \text{elim-base } (\text{cong}^D \ \text{elim } \text{loop}) \equiv \ell
\end{aligned}$$

Note that the definition is parametrised by the level of the universe into which the eliminator should eliminate (*lsuc* is a successor operation for levels). Note also that the computation rule for *loop* is more complicated than in Section 5; the reason is that the computation rule for *base* does not hold by definition.

We can also write down a corresponding definition that uses paths instead of the arbitrary notion of equality and prove that the two definitions are pointwise equivalent:

$$\begin{aligned}
& \text{Circle}^P : (p : \text{Level}) \rightarrow \text{Type } (\text{lsuc } p) \\
& \text{Circle}^P \simeq \text{Circle} : \text{Circle}^P \ p \simeq \text{Circle } p
\end{aligned}$$

Finally we can prove that $\text{Circle}^P \ p$ is inhabited and use the equivalence to derive an implementation of $\text{Circle } p$:

$$\begin{aligned}
& \text{circle}^P : \text{Circle}^P \ p \\
& \text{circle} : \text{Circle } p
\end{aligned}$$

It is even possible to do this in such a way that the derived eliminator gets the “right” computational behaviour for the point constructor (see the accompanying code for details). This implies that the less complicated computation rule given for the higher constructor in Section 5 can be proved.

However, when I followed this method I ended up with quite a bit more code (excluding library code) than when I used the approach demonstrated above. Here is an implementation of $circle^P$, with the proof of the second computation rule omitted:

$$\mathbb{S}^1, \text{base}, \text{loop}^P, \lambda P b \ell \rightarrow \text{elim}^P P b (\text{subst} \Rightarrow \text{Path}^H \{P = P\} \ell), \text{refl}^P b, \dots$$

The code uses a variant of $\text{subst} \Rightarrow \text{Path}^H$ for paths, and the omitted proof of the second computation rule uses a variant of *dependent-computation-rule-lemma* for paths, plus an extra lemma (due to the more complicated formulation of the computation rule). If we do not count the size of library code then this code ($Circle^P$ and $circle^P$) is already about as large as the definition of the eliminator and computation rule given in Section 5. In addition we have to prove $Circle^P \simeq Circle$, and if we are not careful we can end up with an eliminator that does not have the right computational behaviour for the point constructor. Finally we can write a little more code to establish the less complicated formulation of the computation rule for the higher constructor.

9 Discussion

The approach used for suspensions in Section 7 works for several other higher inductive types from the HoTT book [7], including the interval, pushouts, and a general truncation operator (see the accompanying code for details). Furthermore I have shown how one can handle propositional truncation and set truncation constructors. The higher constructors of these types—with the exception of the truncation constructors—satisfy the following two properties:

- All constructors return paths between points.
- No constructor takes a path involving the type family that is being defined as input (ignoring the possibility of later instantiating parameters to such paths).

This might seem to be a serious limitation. However, the HoTT book mentions a method for avoiding paths as inputs [7, Section 6.9], involving the use of auxiliary higher inductive types. It also discusses a method for avoiding higher constructors that return paths between paths, using a “hub” and “spokes” [7, Section 6.7]. The hub-and-spokes construction is used to define the general truncation operator. A potential drawback of the hub-and-spokes construction is that the resulting eliminators may have different computational behaviour [7, Remark 6.7.2], but that is already the case for the methods discussed here.

If the computational behaviour for higher constructors is important, then I do not advocate using the techniques discussed above. Working directly with paths can lead to better computational behaviour: the J rule might not compute in the usual way, but instead there are new definitional equalities. To illustrate this point: The HoTT book mentions another higher inductive type, the torus, given using the hub-and-spokes construction. The definition refers to the circle. I could use a variation of the method described in this text to give an interface to the torus. However, when doing this I found it easier to work with the path-based interface to the circle than the one using an arbitrary notion of equality. (For details of my definition, see the accompanying code.)

References

- 1 The Agda Team. *Agda User Manual, Release 2.6.1*, 2020. URL: <https://readthedocs.org/projects/agda/downloads/pdf/v2.6.1/>.
- 2 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs, TYPES 2015*, number 69 in LIPIcs, page 5:1–5:34, 2018. doi:10.4230/LIPIcs.TYPES.2015.5.

10:18 Higher Inductive Type Eliminators Without Paths

- 3 Nils Anders Danielsson. Code related to the paper “Higher Inductive Type Eliminators Without Paths”, 2020. doi:10.5281/zenodo.3941063.
- 4 Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five Years of Constructive Type Theory: Proceedings of a Congress Held in Venice, October 1995*, volume 36 of *Oxford Logic Guides*, page 83–111. Oxford University Press, 1998.
- 5 Anders Mörtberg, Andrea Vezzosi, et al. An experimental library for Cubical Agda. Agda code, 2020. URL: <https://github.com/agda/cubical/>.
- 6 Andrew Swan. An algebraic weak factorisation system on 01-substitution sets: A constructive proof. *Journal of Logic & Analysis*, 8(1):1–35, 2016. doi:10.4115/jla.2016.8.1.
- 7 The Univalent Foundations Program. Homotopy type theory, 2013. First edition. URL: <https://homotopytypetheory.org/book/>.
- 8 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):87:1–87:29, 2019. doi:10.1145/3341691.