18th International Workshop on Types for Proofs and Programs

TYPES 2011, September 8–11, 2011, Bergen, Norway

Edited by

Nils Anders Danielsson
Bengt Nordström

Editors

Nils Anders Danielsson, Bengt Nordström
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
nad@chalmers.se, bengt@chalmers.se

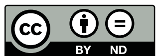
ACM Classification 1998
D.1.1 Applicative (Functional) Programming, D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.1 Mathematical Logic

ISBN 978-3-939897-49-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-49-1.

Publication date
January, 2013

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 .

License

This work is licensed under a Creative Commons Attribution-NoDerivs 3.0 Unported license: http://creativecommons.org/licenses/by-nd/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.
- No derivation: It is not allowed to alter or transform this work.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2011.i

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

- Susanne Albers (Humboldt University Berlin)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Wolfgang Thomas (RWTH Aachen)
- Vinay V. (Chennai Mathematical Institute)
- Pascal Weil (*Chair*, University Bordeaux)
- Reinhard Wilhelm (Saarland University, Schloss Dagstuhl)

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

Preface vii

List of Authors ix

Non-constructive complex analysis in Coq
Aloïs Brunel 1

Infinitary Rewriting Coinductively
Jörg Endrullis and Andrew Polonsky 16

A new approach to the semantics of model diagrams
Johan G. Granström 28

Testing versus proving in climate impact research
Cezar Ionescu and Patrik Jansson 41

Verification of redecoration for infinite triangular matrices using coinduction
Ralph Matthes and Celia Picard 55

18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson and Bengt Nordström

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

Preface

The 18th International Workshop on Types for Proofs and Programs was held in Bergen, Norway from September 8 to September 11, 2011. It was attended by 130 researchers. The local organisers were Marc Bezem and Michał Walicki and the program committee consisted of Marc Bezem from University of Bergen and Ana Bove, Thierry Coquand, Nils Anders Danielsson, Peter Dybjer and Bengt Nordström (chair) from Chalmers University of Technology and University of Gothenburg.

The TYPES workshops were first organised in the late 1980's and were supported by a series of EU programmes from 1989 to 2008. Previous workshops were held in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal (2002), Turin (2003), Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Turin (2008), Aussois (2009) and Warsaw (2010).

There were 40 presentations at the workshop and 8 submissions to these open post-proceedings, out of which 5 papers were accepted. Three of the papers represent theoretical advances in type theory; two describe interdisciplinary applications: model-driven engineering and climate impact research.

We wish to thank the people who served on the programme committee for these proceedings: Andreas Abel (Ludwig-Maximilians-Universität München), Hugo Herbelin (INRIA Paris-Rocquencourt), Zhaohui Luo (Royal Holloway, University of London), Claudio Sacerdoti Coen (University of Bologna) and Tarmo Uustalu (Tallinn University of Technology). We also wish to thank the following additional reviewers: Robin Adams, Ferruccio Guidi, Stefan Kahrs, Marco Maggesi, Conor McBride, Mauro Piccolo, Enrico Tassi and Tao Xue.

December 2012 Nils Anders Danielsson, Bengt Nordström

18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson and Bengt Nordström

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

List of Authors

Aloïs Brunel
Université Paris 13, Sorbonne Paris Cité, Laboratoire d'Informatique de Paris-Nord (LIPN), CNRS, UMR 7030
F-93430, Villetaneuse, France
alois.brunel@ens-lyon.org

Jörg Endrullis
Department of Computer Science, VU University Amsterdam
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
j.endrullis@vu.nl

Johan G. Granström
Google
Brandschenkestrasse 110
8002 Zürich, Switzerland
georg.granstrom@acm.org
Department of Computer Science
King's College London
Strand, London, WC2R 2LS, U.K.

Cezar Ionescu
Potsdam Institute for Climate Impact Research
Telegrafenberg A31, 14473 Potsdam, Germany
ionescu@pik-potsdam.de

Patrik Jansson
CSE Department, Chalmers University of Technology
SE - 412 96 Göteborg, Sweden
patrikj@chalmers.se

Ralph Matthes
Institut de Recherche en Informatique de Toulouse (IRIT)
C.N.R.S. and University of Toulouse, France

Celia Picard
Institut de Recherche en Informatique de Toulouse (IRIT)
University of Toulouse, France

Andrew Polonsky
Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
andrew.polonsky@gmail.com

18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson and Bengt Nordström

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

Non-constructive complex analysis in Coq

Aloïs Brunel

Université Paris 13, Sorbonne Paris Cité, Laboratoire d'Informatique de Paris-Nord (LIPN), CNRS, UMR 7030, F-93430, Villetaneuse, France.
alois.brunel@ens-lyon.org

Abstract

Winding numbers are fundamental objects arising in algebraic topology, with many applications in non-constructive complex analysis. We present a formalization in Coq of the winding numbers and their main properties. As an application of this development, we also give non-constructive proofs of the following theorems: the Fundamental Theorem of Algebra, the 2-dimensional Brouwer Fixed-Point theorem and the 2-dimensional Borsuk-Ulam theorem.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Coq, complex analysis, winding numbers

Digital Object Identifier 10.4230/LIPIcs.TYPES.2011.1

1 Introduction

In this paper we present a formalization in Coq of several results in complex analysis. More precisely, we have formalized non-constructive proofs of the following theorems: the two-dimensional Brouwer Fixed Point theorem, the two-dimensional Borsuk-Ulam theorem and the Fundamental Theorem of Algebra. The particularity of these proofs, besides their classical nature (in the logical sense), is that they all rely on the notion of winding number, which is an invariant of homotopy. The winding number around a point $z \in \mathbb{C}$ of a closed curve γ basically counts how many time γ *turns* counterclockwise around z. They constitute an important notion in algebraic topology and have applications in many domains of mathematics and physics, including complex analysis but also differential geometry and string theory. This wide range of applications has decided us to start the formalization of this notion in Coq, along with examples of important applications.

Finally, we are also interested in organizing our development in a reusable set of libraries on top of Coq Standard Library. There is still some cleaning and organizing work to do on our development, but we think the presented work is close to that goal.

Contributions

To establish these results, we had to develop a whole library on top of the Coq Standard Library. It includes a general purpose library for metric spaces, defined using type classes [12], that generalize several results of the Coq Standard Library of reals. We have formalized some properties of Euclidean spaces, including the characterization of compact sets as the bounded closed sets. Our formalization also provides definitions and various results about the complex plane: the definition and the continuity of common functions, the existence of a complex logarithm and a continuous lifting theorem. Finally, a crucial part of the formalization concerns the definition of the winding number of a closed path and its main properties, culminating in proving that the winding number is an homotopy invariant. Results about line integrals have also been formalised but they are just briefly discussed in this paper. To

© Aloïs Brunel;
licensed under Creative Commons License BY-ND
18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson, Bengt Nordström; pp. 1–15

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

define winding numbers, we have followed [11]. The proofs of the three mentioned theorems we have formalised can be found in [5].

The classical nature of the proofs presented here is twofold. First, we decisively use the reasoning by contradiction to obtain the different results from the homotopy invariance theorem (although it is not used to define winding numbers or to prove the homotopy invariance of the winding number). Secondly, the standard library of reals of Coq is a classical axiomatization of the field of reals.

Related work

C-CoRN Project — A constructive proof of the Fundamental Theorem of Algebra has been formalized as part as the C-CoRN project [2]. Its proof [6] relies on elementary properties of $\mathbb{R}$ and $\mathbb{C}$ (mostly the existence of k-th roots in $\mathbb{C}$, an intermediate value theorem for polynomials and some basic polynomial arithmetic). The constructive nature and the careful design of the proof makes it particularly suitable for extraction [3]. In contrast, we were interested in formalizing classical mathematics, which makes our two works completely different in nature. Yet, it does not mean we completely give up on the possibility of extraction, as discussed in the conclusion.

Coqtail Project — Coqtail [4] is a project intended to extend the standard Coq library by providing clean, reusable libraries for various domains of undergraduate mathematics: arithmetic, reals, basic complex analysis, basic topology. It has been used to formalize a proof of Lagrange's four square theorem, to formalize power series and solve some differential equations [1]. It seems that many of the basic definitions about complex numbers and functions coincide in both our works, and so it is likely that the developments described here could easily be integrated in their library.

Other proof assistants — Numerous developments based on complex analysis, euclidean spaces or topology have been formalized in other proof assistants. One can cite Harrison's works in HOL Light [8, 7] on the theory of Euclidean spaces (including a proof of the general Brouwer Fixed-Point theorem, using combinatorial arguments) and on a complex-analytic proof of the prime number theorem.

Outline

Basic definitions and notations are described in section 2. We then present in section 3 the metric spaces and euclidean spaces libraries. Section 4 introduces the existence of a complex logarithm, the continuous lifting theorem and finally the definition of the winding number and the formalization of some of its main properties. We present the non-constructive proof of the main theorems along with their formalization in section 5. Section 6 finally concludes this work.

2 Basic notations and definitions

We give here the basic notations and definitions, relative to the complex plane and the euclidean spaces in general, needed to understand the Coq statements of the next sections.

2.1 Complex plane

We begin with the definition of the complex plane. We remind that the underlying theory of reals that we use is the one of the Coq Standard Library. It is based on an axiomatic definition of the field of reals. It has to be noted that this axiomatic definition of reals is fundamentally classical.

The set $\mathbb{C}$ is defined as $\mathbb{R}^2$, the imaginary and real parts being respectively the first and second projections.

```
Definition C : Set := prod R R.
Definition CRe (c : C) : R := match c with ( a, _) => a end.
Definition CIm (c : C) : R := match c with ( _, b) => b end.
```

`co : R*R -> C` denotes the trivial coercion from `R*R` to `C`, so that `co a b` represents the complex number $a + ib$. A coercion from `R` to `C` is defined and noted `IRC`.

```
Definition IRC (r : R) : C := co r 0.
Coercion IRC : R >-> C.
```

It also possible to define a complex number by its polar form.

```
Definition polar_form (r: R) (t : R) :=  co (r*cos t) (r*sin t).
```

We declare distinguished elements of `C`, noted `0`, `1`, `Ci` representing respectively 0, 1 and the purely imaginary number i. We endow `C` with operations noted as in `R`, `+`, `-`, `*` and `/`. `C` then defines a ring and a field, and is declared as such, permitting to use the `ring` and `field` tactic families. The absolut value and the conjugation operations are noted respectively `Cmod` and `Cconj`. The complex exponential is defined using the real exponential already defined in the standard library and the polar form:

```
Definition Cexp (c : C) := polar_form (exp (CRe c)) (CIm c).
```

The circle of radius `r` can then be parametrized as follows.

```
Definition C_circle_par (r : R) (theta : R) : C :=  r * Cexp (0,theta).
```

2.2 Euclidean space

We also define the euclidean space $\mathbb{R}^n$, using an inductive product of set.

```
Fixpoint prod_n (E: Set) (n:nat) : Set :=
  match n with
   | O => unit
   | S n => prod (prod_n E n) E
  end.
```

Hence `prod_n R n` represents the set $\mathbb{R}^n$. We define usual operations on $\mathbb{R}^n$: `[+]`, `[-]`, `[.]` and an element `[0]` implementing respectively the addition, the substraction, the inner product and the element $(0, \ldots, 0)$.

2.3 Domains

We define useful subsets of $\mathbb{R}$ and $\mathbb{C}$.

```
Definition CUnit_Disk : C -> Prop := fun x => Cmod x <= 1.
Definition CUnit_Circle : C -> Prop := fun x => Cmod x = 1.
Definition RDom_Int (a b : R) : R -> Prop := fun x => a <= x <= b.
Definition CRect (a b c d : R) : C -> Prop := fun c => a <= CRe c <= b
/\ d <= CIm c <= e.
```

We define the notion of star shaped subset of $\mathbb{C}$.

```
Definition CDom_Star (K : C -> Prop) :=
  forall x y : C, K x -> K y ->
  forall lam : R, 0 <= lam <= 1 -> K (lam * x + (1 - IRC lam) * y).
```

3 Metric spaces

To prove sophisticated complex analysis results, we need elementary regularity properties of functions on $\mathbb{R}^n$, which are consequences of the metric space structure of $\mathbb{R}^n$, such as Heine theorem (continuity on a compact implies uniform continuity). There are also properties we need for $\mathbb{R}, \mathbb{R}^2$ and $\mathbb{R}^3$, which are true for all euclidean spaces $\mathbb{R}^n$. Instead of reproving these results each time we consider a different set, we do it in the general case. Hence, we provide libraries for metric spaces and euclidean spaces, which are presented in this section.

3.1 Metric spaces

We define metric spaces and of top of them, the notions of continuity, uniform continuity, open set, closed set, and so on. We mostly follow the definitions and naming already present in Coq reals library. Our choice has been to define metric spaces as a type class [12], hence benefiting of features like notation overloading, parametrized instances and generalized type-class binders. The definition is as follows:

```
Class MetrSpace (E: Set) :=
{
  d : E -> E -> R;
  pr_pos : forall x y : E, 0 <= d x y;
  pr_sym : forall x y : E, (d x y) = (d y x);
  pr_sep : forall x y : E, (d x y = 0) <-> x = y;
  pr_tri : forall x y z : E, d x y <= (d x z) + (d z y);
  pt   : E
}.
```

► Remark. Notice that we define *pointed* metric spaces, that is a metric space together with a distinguished element `pt` of the base set. This is just a convenient choice that simplifies a bit some proofs abound bounded sets and the writing of some tactics about continuity (that we don't mention in this paper).

Different instances of the class `MetrSpace` are declared, like $\mathbb{R}$ and $\mathbb{C}$. We also define the product metric space of two metric spaces as a parametrized instance:

```
Instance prod_MetrSpace `(EM : MetrSpace E, FM: MetrSpace F) :
                                   MetrSpace (E * F).
```

This allows to declare the n-dimensional euclidean space $\mathbb{R}^n$ as a metric space.

In the definition of limits and continuity between two metric spaces, the latter are introduced via generalized type-class binders, which allows to write statements and proofs in a natural way.

```
Definition Metr_limit_in `{EM: MetrSpace E, FM: MetrSpace F}
  (f : E -> F) (D : E -> Prop) (a : E) (l : F) :=
  forall eps : posreal -> exists eta : posreal /\
   (forall x, D x -> d x a < eta -> d (f x) l < eps).

Definition MS_continue_in `{EM : MetrSpace E, FM : MetrSpace F}
  (f : E -> F) (D : E -> Prop) (a : E) : Prop :=
    Metr_limit_in f D a (f a).

Definition MS_uniform_continuity `{EM : MetrSpace E, FM : MetrSpace F}
  (f : E -> F) (D : E -> Prop) : Prop :=
  forall eps : posreal, exists delta : posreal,
      (forall x y: E, D x -> D y ->
       d x y < delta -> d (f x) (f y) < eps).
```

▶ **Example 1.** As an example, the statement that the function $x \in \mathbb{R} \mapsto x + 1$ is uniformly continuous on $\mathbb{R}$ is simply written in Coq:

```
MS_uniform_continuity (fun x => x + R1) (fun x => True)
```

The notation is light: there is no need to specify the base set nor the metric space used here, since the type-class constraint system permits to retrieve the previously declared metric space on $\mathbb{R}$.

The definition of compact set is adapted from the one used in the Coq Reals library. It is however a notion of compactness with respect to a set of open sets $\mathcal{O}$.

```
Definition MS_compact_base `{EM: MetrSpace E}
  (X: E -> Prop) (O: (E->Prop)->Prop) : Prop :=
  forall I : Type, forall IM: MetrSpace I, forall f : MS_family I E,
    MS_covering_open_set X f -> MS_family_base f O ->
     exists D : I -> Prop, MS_covering_finite X (MS_subfamily f D).
```

This amounts to say that a set X is compact if whenever we have a cover $\mathcal{C}$ of X constituted by open sets of $\mathcal{O}$, we can find a finite subset $\mathcal{C}' \subseteq \mathcal{C}$ which is still a cover of X. The usual compactness property is just an alias for compactness with respect to all open sets.

```
Definition MS_compact `{EM: MetrSpace E}  (X: E -> Prop) : Prop :=
  MS_compact_base X (fun _ => True).
```

In the Real library, a cover is represented by a family of open sets $(O_i)_{i\in\mathbb{R}}$ indexed by $\mathbb{R}$. Here, we can use any element of `Type` as a set of indexes. This is indeed necessary to prove a crucial result: compactness is equivalent to compactness with respect to an open set basis.

```
Theorem MS_compact_basis `{EM : MetrSpace E}:
  forall X : E -> Prop, forall O : (E -> Prop) -> Prop,
  forall Ho: MS_open_basis O, MS_compact_base O X -> MS_compact X.
```

Here, `MS_open_basis O` denotes the fact that a set $\mathcal{O}$ of open sets is such that any open set G can be written $G = \bigcup_{X \in O \wedge X \subseteq G} X$. To prove this theorem, we *need* to have an index set of type `Type`. Indeed, from an original cover C_i, we build the cover $C_{(i,U)}$ where U is an open set such that $U \subseteq C_i$ and $U \in \mathcal{O}$. This amounts to use `prod I (E -> Prop)` as a type for indexes, which justifies the use of `Type`.

▶ **Example 2.** As an example of a theorem already proved for $\mathbb{R}$ in the standard library, the Heine theorem is now available for all metric spaces. It states that every continuous function on a compact set is also uniformly continuous.

```
Theorem MS_Heine :
  forall (f:E -> F) (D:E -> Prop),
    MS_compact D -> MS_continue_on f D -> MS_uniform_continuity f X.
```

3.2 Euclidean spaces

Rather than defining directly euclidean spaces with the particular canonical euclidean scalar product, we define them axiomatically as a type class:

```
Class Euclidean (dim : nat) :=
{
  scal : prod_n R dim -> prod_n R dim -> R;
  scal_sym : forall x y : prod_n R dim, scal x y = scal y x;
  scal_pos : forall x : prod_n R dim, 0 <= scal x x;
  scal_def : forall x  : prod_n R dim, scal x x = R0 -> x = Rn_zero;
  scal_add1 : forall x y z : prod_n R dim, scal (Rn_plus x y) z =
                      scal x z + scal y z;
  scal_add2 : forall x y z : prod_n R dim, scal x (Rn_plus y z)  =
                      scal x y + scal x z;
  scal_lam1 : forall x y lam, scal (Rn_dot  x lam) y= lam * scal x y;
  scal_lam2 : forall x y lam, scal x (Rn_dot  y lam)= lam * scal x y
}.
```

Each instance of an euclidean space then defines an euclidean norm, defined as follows:

```
Definition Eucl_norm `{E : Euclidean n} :=
  fun x : prod_n R n => sqrt (scal x x).
```

We define two notations for the unit disk and the unit circle of dimension n.

```
Definition RnUnit_Disk `{E : Euclidean n} :=
    fun x : prod_n R n => Eucl_norm x <= 1.
Definition RnUnit_Circle `{E : Euclidean n} :=
    fun x : prod_n R n => Eucl_norm x = 1.
```

From these axioms, we derive several useful properties, like the Cauchy-Schwarz inequality.

```
Lemma Eucl_CauchySchwartz `{E: Euclidean n}:
 forall x y, Rabs (scal x y) <= Eucl_norm x * Eucl_norm y.
```

Using the euclidean norm to define a distance, we can show that each euclidean space $\mathbb{R}^n$ defines a metric space instance.

```
Instance Rn_MetrSpace {n : nat} : MetrSpace (prod_n R n).
```

An important step in our development is the Borel-Lebesgue theorem, which states that in $\mathbb{R}^n$, the compact sets (defined in terms of covering) are exactly those sets which are both closed and bounded.

```
Theorem Eucl_Borel_Lebesgue:
  forall n : nat, forall X : prod_n R n -> Prop,
    (MS_compact X <-> MS_closed_set X /\ MS_bounded X).
```

In particular, to show that a closed and bounded set is compact, we reason by induction and use the fact that a product of compacts is compact. That is where we need the equivalence between compactness and compactness on the product basis (which, for the product metric space, is the set of product of open sets) stated in the previous subsection.

4 Winding number theory

There are many ways to define the winding number. Mostly, two approaches are possible: by using path integral or by proving a lifting theorem. We have formalised both definitions, but we focus only on the latter, since it is more general and presents many advantages, as advocated in subsection 4.4 In this section, we present the following results: the existence of a complex logarithm, a continuous lifting theorem, and finally the notion of winding numbers.

4.1 Complex logarithm

A complex logarithm is an *inverse* of the complex exponential function, similarly to the case of the real-valued functions ln and e^x. However, the situation is more complicated on $\mathbb{C}$ than on $\mathbb{R}$. Indeed, the complex exponential is not injective (just consider the identity $e^x = e^{x+2i\pi}$) and hence cannot have an inverse function. This problem is usually solved by restricting the domain of the exponential to a subset on which it is injective. In our case, we restrict it to $\mathbb{R}\times]-\pi,\pi]$, and hence the logarithm will be defined only on the domain $\mathbb{C}\setminus\mathbb{R}_-$, which is defined in Coq as:

```
Definition CLog_D0 := fun c => forall x : R, x <= 0 -> c <> IRC x.
```

We first show that every point z of this domain has a logarithm. To prove that it suffices to notice that by the domain restriction, the polar decomposition of $z = re^{i\theta}$ is unique. This fact is equivalent to the following statement.

```
Lemma CLog_1:
  forall z, CLog_D0 z ->
    exists r, exists theta,  0 < r /\ -PI < theta <= PI /\
      (IRC r) * Cexp (co 0 theta) = z.
```

Hence, its logarithm can be defined by $Log(z) = ln(r) + i\theta$. We can then prove the *existence* of a logarithm function on the domain `CLog_D0`. This function is necessarily continuous.

```
Lemma CLog_ex_continuous :
  exists log : C -> C, log C1  = C0 /\
  (forall z, CLog_D0 z ->
     -PI <= CIm (log z) <= PI /\ Cexp (log z) = z /\ MS_continue_in log z).
```

To prove this theorem, we crucially need the axiom of choice in its functional form:

```
Axiom choice :
 forall (A B : Type) (R : A->B->Prop),
   (forall x : A, exists y : B, R x y) ->
    exists f : A->B, (forall x : A, R x (f x)).
```

We could obtain an actual function `log : C -> C` by using the principle of constructive indefinite description.

```
Axiom constructive_indefinite_description :
  forall (A : Type) (P : A->Prop),
    (exists x, P x) -> { x : A | P x }.
```

This principle is stronger than the axiom of choice. In fact, we never need to obtain a logarithm function: the statement of its existence is enough.

4.2 Complex lifting

Given a function $f : \mathbb{C} \to \mathbb{C}$ continuous on K, we say that $\Phi : K \to \mathbb{C}$ is a continuous lifting of f if Φ is continuous and $\forall x \in K, f(x) = \|f(x)\|e^{\Phi(x)}$. We can state the existence of such a lifting for any set K, which is both compact and star-shaped.

```
Theorem Complex_Lifting:
  forall F : C -> C, forall K : C -> Prop,
  MS_compact K ->  CDom_Star K -> MS_continue_on F K ->
  (forall x, K x -> F x <> C0) -> exists Phi : C -> C,
      (forall x : C, K x -> F x = IRC (Cmod (F x)) * Cexp (Phi x)) /\
      MS_continue_on Phi K.
```

The proof, which we don't detail, crucially relies on the uniform continuity of the function, and hence on Heine theorem.

4.3 Winding numbers

A *path* is a continuous function $\gamma : [a, b] \to \mathbb{C}$. We moreover say it is a *closed path* if $\gamma(a) = \gamma(b)$. From now on, we only consider closed path γ such that $\forall x \in [a, b], \gamma(x) \neq 0$. In Coq, a closed path is represented as a record containing its domain together with a proof of its continuity.

```
Record C_lace : Type := mklace {
  gam :> R -> C;
  a : R;
  b : R;
  ab_pr: a <= b;
  gam_lace : gam a = gam b;
  gam_cont: forall x, RDom_Int a b x -> MS_continue_in gam (RDom_Int a b) x
}.
```

Given a closed path g, we say that $\psi : [a, b] \to \mathbb{C}$ is an argument of g if ψ is continuous and if $\forall x \in [a, b], g(x) = \|g(x)\|e^{\psi(x)}$. This property is denoted in Coq by

```
Definition cont_arg_choice (a b : R) (F : R -> C) (psi : R -> C) :=
   (forall x, RDom_Int a b x -> F x = IRC(Cmod(F x))*(Cexp (psi x)))
/\ (forall x, RDom_Int a b x -> MS_continue_in psi (RDom_Int a b) x).
```

If γ is nowhere vanishing (meaning it never takes the value 0 on its domain) and H is an argument of γ, we can define its *winding number* (around 0) by:

```
Definition lace_WN_param (g : C_lace) (psi : R -> C) : C :=
  (psi (b g) - psi (a g))/(co 0 (2*PI)).
```

Moreover, we prove that whatever the choice of argument we have made, the winding number is the same.

```
Lemma lace_WN_param_equal:
  forall g, (forall x, RDom_Int (a g) (b g) x -> g x <> C0) ->
  forall psi1 psi2,
    cont_arg_choice (a g) (b g) g psi1 ->
    cont_arg_choice (a g) (b g) g psi2 ->
       lace_WN_param g psi1  = lace_WN_param g psi2.
```

▶ Remark. Usually, because the winding number is invariant by the choice of argument, it is defined as an actual number using a specific continuous argument ψ obtained by the complex lifting theorem.

$$n(\gamma,0)=\frac{\psi(b)-\psi(a)}{2i\pi}$$

We choose not to do that since it would mean using the principle of constructive indefinite description to obtain an argument, which can be avoided. Instead, we will always carry an assumption of the existence of a continuous argument.

An important property is that the winding number of a closed path is always an integer.

```
Lemma lace_WN_param_Z:
  forall g psi,
    (forall x, RDom_Int (a g) (b g) x -> g x <> C0) ->
    cont_arg_choice (a g) (b g) g psi ->
    exists z : Z, lace_WN_param g psi = IRC (IZR z).
```

To obtain this result, we make use of trigonometry results contained in the standard library. Here is an informal proof.

Proof. Suppose that for every x, $\gamma(x)$ is in the unit disk. Let Φ be a lifting of γ: $\gamma(x) = |\gamma(x)|e^{\Phi(x)}$. Then, $e^{\Phi(b)-\Phi(a)} = 1$ (since $\gamma(a) = \gamma(b)$. Hence, there exists some $k \in \mathbb{Z}$ such that $\Phi(b) - \Phi(a) = 2i(k\pi)$. Hence $n(\gamma, 0) = \frac{\Phi(b)-\Phi(a)}{2i\pi} = k \in \mathbb{Z}$. ◀

▶ **Example 3.** As an example, we can compute the winding number of the unit circle.

```
Definition C_circ_unit : R -> C := fun t => Cexp (co 0 (2*PI*t)).
```

The winding number of the corresponding path `C_circ_lace` between 0 and 1 is equal to 1. This fits the intuition of the path turning one time around the point 0.

```
Lemma C_circ_fact2:
  forall psi, cont_arg_choice 0  1 (C_circ_lace) psi ->
    lace_WN_param C_circ_lace psi  = C1.
```

The final and important theorem is the *invariance of the winding number by homotopy*. Formally, supposing two closed paths `g0, g1 : C_lace` are *homotopically equivalent*, that is there exists a continuous function `H : C -> C` such that:

```
Definition CHomotopyEqu (g0 g1 : C_lace) (H : C -> C) :=
  a g0 = a g1 /\ b g0 = b g1 /\
  (forall x, a g0 <= x <= b g0 -> H(0,x) = g0 x) /\
  (forall x, a g1 <= x <= b g1 -> H(1,x) = g1 x) /\
  (MS_continue_on H (CRect 0 1 (a g0) (b g0))) /\
  (forall x, RDom_Int 0 1 x -> H(u, a g0) = H(u, b g0)).
```

And if moreover, `H` never equals to `0` (which ensures that neither `g0` nor `g1` do), then the winding numbers of `g0` and `g1` are equal. This is summarized in the following theorem:

```
Theorem Clace_WN_homotopy_invariant:
  forall g0 g1 : C_lace, forall H : C -> C,
  (forall c, (CRect 0 1 (a g0) (b g0) c) -> H c <> C0) ->
  CHomotopyEqu g0 g1 H ->
  forall psi0 psi1 : R -> C,
    cont_arg_choice (a g0) (b g0) g0 psi0 ->
    cont_arg_choice (a g1) (b g1) g1 psi1 ->
      lace_WN_param g0 psi0 = lace_WN_param g1 psi1.
```

▶ Remark. Notice that here again, the theorem is stated without fixing a choice of argument for the closed paths.

4.4 Winding numbers: path integral versus continuous lifting

We have presented here a definition of winding number of a closed path by using a choice of argument for it. It is however often defined using line integrals. We can indeed define the winding number of a closed path $\gamma : [a, b] \to \mathbb{C}$ around a point c as:

$$n(\gamma, c) = \frac{1}{2i\pi} \oint_\gamma \frac{dz}{z - c}$$

where the line integral is defined using

$$\oint_\gamma f(z)dz = \int_a^b f(\gamma(t))\gamma'(t)dt$$

We have also formalized this alternative definition and proved that it yields the same result as the other. It has shown several disadvantages over the definition we have presented:

- To define path integrals, we need a good definition of integration for complex valued functions over $\mathbb{R}$. We have experimented using the Riemann integral from the Standard Library of Coq. It allows one to define winding numbers without the path lifting theorem, but always reasoning on integrals rather than in terms of complex exponentials and logarithms is definitely more difficult.
- The main problem is that because we use path integrals, we also need the path γ to be differentiable (it can be then extended for continuous paths, but it involves sophisticated results about complex analysis we have not formalised). This is indeed a severe restriction, since we could prove the Fundamental Theorem of Algebra, but not the Brouwer Fixed-Point theorem or the Borsuk-Ulam theorem, which are stated for continuous functions. In contrast, coupled with the continuous lifting theorem, our definition immediately only requires continuity of γ.

5 Applications of the winding number homotopy invariance

We now detail the proofs we have formalized of the Fundamental Theorem of Algebra, the Brouwer Fixed-Point theorem and finally of the Borsuk-Ulam theorem. All these proofs rely on corollaries of the invariance by homotopy of the winding number and classical principles.

5.1 Prerequisites

We briefly give the statements and sketch the proofs of two fundamental lemmas needed for the proofs of the Borsuk-Ulam and Brouwer Fixed Point theorems.

▶ **Lemma 4.** *Suppose $f : \mathbb{C} \to \mathbb{C}$ is continuous and nowhere vanishing on the unit disk. Then if $\gamma(t) = f(e^{2i\pi t})$, we have $n(\gamma, 0) = 0$.*

Proof. The path γ is homotopically equivalent to the constant path $t \mapsto f(0)$. Indeed, $H(u,t) = f(u * e^{2i\pi t})$ is such that $H(0,t) = f(0)$ and $H(1,t) = \gamma(t)$. It is moreover continuous because f is, and vanishes nowhere. Hence, because any constant path has a winding number equal to 0, we conclude by homotopy invariance of the winding number. ◀

▶ **Lemma 5.** *There does not exist a map $f : \mathbb{C} \to \mathbb{C}$ which is continuous, odd (that is $f(-x) = -f(x)$) and nowhere vanishing on the unit disk.*

Proof. We will prove that if such a map f exists, then if we pose the lace $\gamma(t) = f(e^{i\pi t})$, there exists $k \in \mathbb{Z}$ such that $n(\gamma, 0) = 2k + 1$ (we skip the proof here, but it only involves simple calculations). Hence, because of Lemma 4, it leads to a contradiction. ◀

5.2 Fundamental Theorem of Algebra

The first application is a classical proof of the Fundamental Theorem of Algebra, which states that any complex polynomial has a root. A complex polynomial is represented as a list of complex coefficients, begining with the coefficient of higher degree and ending with the one of degree 0.

```
Definition C_polynom : Set := Clist.
Definition C_polynom_deg (P : C_polynom) := pred (Clength P).
```

But of course, we need to remove the extra elements equals to `C0` in order to be able to calculate the true degree of the polynomial. This is the job of the function `C_polynom_without_zero` which has the type `C_polynom -> C_polynom`. The evaluation of a polynomial is done inductively by the function `C_polynom_eval : C_polynom -> C`. We now prove the following statement.

```
Theorem FTA: forall P : C_polynom,
   (1 <= C_polynom_deg (C_polynom_without_zero a)) ->
   exists x : C, C_polynom_eval P x = C0.
```

So suppose the existence of a polynomial `P` of degree `n` (and we note its dominating coefficient $a_n \neq 0$) such that

```
Variable pr_deg : n >= 1.
Variable pr_root: forall x, C_polynom_eval a x <> C0.
```

We then define the lace `Gamma_circle r` whose underlying function is the parametrization of the circle of radius `r` deformed by the polynomial `P` (and by hypothesis `pr_root`, it makes sense to speak of its winding number):

```
fun theta : R => C_polynom_eval P (C_circle_par r theta)
```

Now if R1 is big enough, the polynomial becomes dominated by its coefficient of larger degree `C_polynom_domcoeff P`, and then the winding number is the same wether or not you consider the other coefficients:

```
Definition nu (r : R) (theta : R) : C :=
  C_polynom_domcoeff P * IRC (r^n) * Cexp (co 0 ((INR n)*theta)).

Lemma Alembert_theo5:
  exists M : R, 0 < M /\ forall R1,  forall pr : 0 < R1,
  forall pr2 : M < R1, forall psi1 psi2 : R -> C,
  cont_arg_choice 0 2*PI (nu_path R1 pr) psi1 ->
  cont_arg_choice 0 2*PI (Gamma_circle R1) psi2 ->
  lace_WN_param (nu_path R1 pr) psi1 = lace_WN_param (Gamma_circle R1) psi2.
```

But the winding number of $\nu(\theta) = a_n r^n e^{in\theta}$ can be shown by a simple calculation to be equal to n. When the circle is of radius 0, the obtained path `Gamma_circle 0` is constant and hence its winding number is equal to 0. On the other hand, we can show that whatever the positive reals `R1 R2 : R`, the paths `Gamma_circle R1` and `Gamma_circle R2` are homotopically equivalent, and so have the same winding number.

```
Lemma Alembert_theo3:
  forall R1, 0 <= R1 -> forall R2, 0 <= R2 ->
    (forall psi1 psi2 : R -> C,
       cont_arg_choice 0 2*PI (Gamma_circle R1) psi1 ->
       cont_arg_choice 0 2*PI (Gamma_circle R2) psi2 ->
         forall Arg: R -> (R -> C), forall Harg: (forall r,
           Rmin R1 R2 <= r <= Rmax R1 R2 ->
           cont_arg_choice 0 2*PI (Gamma_circle r) (Arg r)),
lace_WN_param (Gamma_circle R1) psi1 = lace_WN_param (Gamma_circle R2) psi2.
```

The contradiction comes immediately, since when going from `0` to a real `R` big enough, the winding number changes from `0` to `n` (by Lemma `Alembert_theo5`). This is contradicted by the previous lemma `Alembert_theo3` and because `1 <= n`.

5.3 Brouwer Fixed-point theorem

We now prove the 2-dimensional version of the celebrated Brouwer Fixed-Point theorem. It is a classical (in the sense of classical reasoning) corollary of the following no retraction theorem.

```
Theorem No_Retraction:
 ~(exists r : C -> C,
   (forall x, CUnit_Disk x -> CUnit_Disk (r x)) /\
   (forall x, CUnit_Circle x -> r x = x)    /\
   (forall x, CUnit_Disk x -> MS_continue_in r CUnit_Disk x)).
```

Proof. Suppose by contradiction that we have such a retraction r. By hypothesis, for every x in the unit disk, $r(x) \neq 0$, and r is continuous. Hence, by Lemma 4, the lace $\gamma : t \mapsto r(e^{2i\pi t})$ is such that $n(\gamma, 0) = 0$. But, $\gamma(t) = e^{2i\pi t}$ since r is the identity on the unit circle. By Lemma 5, however, we have $n(\gamma, 0) \neq 0$, which is contradictory. ◀

We are now able to formalize a proof of the Brouwer Fixed-Point theorem, which is stated as follows.

```
Theorem BrouwerFixedPoint:
  forall f : C -> C, (forall x, CUnit_Disk x -> CUnit_Disk (f x)) ->
    MS_continue_on f CUnit_Disk ->
    exists x, CUnit_Disk x /\ f x = x.
```

The key point is to reason classicaly by supposing the existence of a map which has no fixpoint and build a retract `CUnit_Disk` to `CUnit_Circle` out of it, which will lead to a contradiction by the no retraction theorem.

Proof. The proof is carried using the following classical principle

```
not_all_not_ex: forall P:U->Prop, ~(forall n:U, ~P n) -> exists n:U, P n.
```

We suppose that `f : C -> C` is continuous on the unit disk and has no fixpoint, and derive a contradiction.

```
Hypothesis Br_H1: forall x, CUnit_Disk x -> CUnit_Disk (f x).
Hypothesis Br_H2: forall x, CUnit_Disk x -> MS_continue_in f CUnit_Disk x.
Hypothesis Br_H3: forall x, CUnit_Disk x -> f x <> x .
```

We want to define a continuous retract `brouwer_retract : C -> C` from the unit disk to the circle. Informally, consider a point `z` of the unit disk and its image `f(z)`. Since we have supposed that `f(z) <> z`, we can continue the segment that joins `f(z)` to `z` until it reaches the unit circle. `brouwer_retract z` is this intersection point. Formally, given two distinct points `x0` and `x` of the unit disk, we need to solve the equation

$$(E) \quad x_0 + \lambda(x - x_0) \quad = \quad 1$$

Finding λ amounts to solve a second degree (real) polynomial, which can be done using the standard Coq library. Given a polynomial aX^2+bX+c, if its discriminant b^2-4ac is positive, the two roots (which are possibly equal) are given by `sol_x1 a b c` and `sol_x2 a b c`. We use this to obtain a function `LC_lambda x x0 : x <> x0 -> R` that calculates the λ of Equation (E).

```
Lemma line_circle_intersect (x0 x : C) (H : x <> x0) :
  CUnit_Disk x0 ->
    Cmod (x0 + IRC (LC_lambda x x0 H) * (x - x0)) = 1 /\
    Cmod x = 1 -> LC_lambda x x0 = 1.
```

The map `brouwer_retract` is then defined, and if `z` is in the unit disk (we have a proof `Hunit : CUnit_Disk z`), it is equal to

```
f z + IRC (LC_lambda z (f z) (Br_H3 z Hunit)) * (z - f z)
```

To conclude, we need to show that `brouwer_retract` is indeed a continuous retract, which amounts to prove the three following lemmas. The first one is the continuity of `brouwer_retract` on the unit disk. This proof involves a lot of bureaucracy, since we have to show that `LC_lambda` is continuous on $\mathbb{C}^*$.

```
Lemma Br_retract_continue : MS_continue_on brouwer_retract CUnit_Disk.
```

Secondly, restricted to the circle, `brouwer_retract` is the identity.

```
Lemma Br_retract_circle :
  forall z : C, CUnit_Circle z -> brouwer_retract z = z.
```

And finally `brouwer_retract` is actually a map from the unit disk to the unit circle.

```
Lemma Br_retract_unit :
  forall z : C, CUnit_Disk z -> CUnit_Circle (brouwer_retract z).
```

These two last lemmas are direct consequences of the lemma `line_circle_intersect`. Under these hypothesis, we conclude to a contradiction.

```
Lemma BrouwerNoFix : False.
```

◀

5.4 Borsuk-Ulam theorem

The last application is the Borsuk-Ulam theorem, which states that for any continuous complex-valued function f on the unit sphere, there exists a point x such that $f(x) = f(-x)$.

```
Theorem BorsukUlam:
  forall f : Rcube -> C, MS_continue_on f RnUnit_disk ->
  exists x, RnUnit_disk x /\ f (-x) = f x.
```

The proof of this theorem will be a consequence of the following intermediate lemma.

```
Lemma BU_lemma2:                #(AC)
  forall f : Rcube -> C, MS_continue_on f RnUnit_disk ->
  (forall x, BU_disk x -> f([-]x) = - f(x)) ->
  exists p, RnUnit_circle p /\ f p = 0.
```

Proof. Here again, we reason by contradiction using `not_all_not_ex`. So we suppose having a map f which is odd, continuous and nowhere vanishing. Then consider the following map (where $\mathcal{S}^2$ is the 2-sphere):

$$
\begin{array}{rcl}
\phi_h & : & \mathbb{R}^2 \to \mathcal{S}^2 \\
\phi_h(x,y) & = & (x, y, \sqrt{1 - x^2 - y^2})
\end{array}
$$

Now, it is clear that if we pose $\gamma(t) = f(e^{2i\pi t}, 0)$, then $\gamma(t) = (f \circ \phi_h)(e^{2i\pi t})$. We know by hypothesis that $f \circ \phi_h$ never vanishes on $\mathbb{C}$ and is continuous. Hence, by Lemma 4, we have $n(\gamma, 0) = 0$. But by Lemma 5, because $f \circ \phi_h$ is nowhere vanishing, odd and continuous, we have $n(\gamma, 0) \neq 0$ which is contradictory. ◀

Given this last lemma, we obtain Borsuk-Ulam Theorem.

Proof of Borsuk-Ulam Theorem. We reason classicaly by supposing the existence of a function $f : \mathbb{R}^3 \to \mathbb{C}$, continuous on the unit ball and such that for every point x of the unit ball, $f(x) \neq f(-x)$. Then, consider the map

$$
F(x) = \frac{f(x) - f(-x)}{\|f(x) - f(-x)\|}
$$

Then F is well-defined and continuous by hypothesis, and F is clearly odd. Hence, using `BU_lemma2`, there exists x in the unit sphere such that $F(x) = 0$, which contradicts the hypothesis since it means $f(x) = f(-x)$. ◀

6 Conclusion and remarks

We have described in this paper a library implementing metric spaces, euclidean spaces and winding numbers, and we have employed it to prove sophisticated results in classical complex analysis. One future direction of research is the generalization of the results in arbitrary dimension. In this development, we have only proved the 2-dimensional version of Brouwer Fixed-Point and Borsuk-Ulam theorems, but their n-dimensional versions still can be proved. The proofs are quite similar to those we have briefly sketched here. However, it requires to use a generalization of the notion of winding number: the *degree* of a continuous mapping. It can be defined for maps from $\mathbb{R}^n$ to $\mathbb{R}^n$ (which is sufficient) but also for continuous mapping between oriented compact manifolds of the same dimension. To do this, one would need to formalize some parts of classical homotopy theory.

References

1 G. Allais. Using reflection to solve some differential equations. *3rd Coq workshop*, 2011.
2 Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Mathematical Knowledge Management*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer Berlin Heidelberg, 2004.
3 Luís Cruz-Filipe and Bas Spitters. Program extraction from large proof developments. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 205–220. Springer Berlin Heidelberg, 2003.
4 Coqtail development team. http://coqtail.sourceforge.net.
5 W. Fulton. *Algebraic topology: a first course*. Springer, 1995.
6 Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack, editors, *Types for Proofs and Programs*, volume 2277 of *Lecture Notes in Computer Science*, pages 96–111. Springer Berlin Heidelberg, 2002.
7 J. Harrison. Formalizing an analytic proof of the prime number theorem. *Journal of Automated Reasoning*, 43(3):243–261, 2009.
8 John Harrison. A HOL theory of euclidean space. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2005.
9 J.L. Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009.
10 Alexandre Miquel. Classical program extraction in the calculus of constructions. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327. Springer Berlin Heidelberg, 2007.
11 M. Rao and H. Stetkaer. *Complex analysis: an invitation: a concise introduction to complex function theory*. World Scientific, 1991.
12 Matthieu Sozeau and Nicolas Oury. First-class type classes. In OtmaneAit Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer Berlin Heidelberg, 2008.

Infinitary Rewriting Coinductively

Jörg Endrullis[1] and Andrew Polonsky[2]

1 **Department of Computer Science, VU University Amsterdam**
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands.
`j.endrullis@vu.nl`

2 **Institute for Computing and Information Sciences, Radboud University Nijmegen,**
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
`andrew.polonsky@gmail.com`

Abstract

We provide a coinductive definition of strongly convergent reductions between infinite lambda terms. This approach avoids the notions of ordinals and metric convergence which have appeared in the earlier definitions of the concept. As an illustration, we prove the existence part of the infinitary standardization theorem. The proof is fully formalized in Coq using coinductive types. The paper concludes with a characterization of infinite lambda terms which reduce to themselves in a single beta step.

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, D.3.1 Formal Definitions and Theory, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems, I.1.1 Expressions and Their Representation, I.1.3 Languages and Systems

Keywords and phrases infinitary rewriting, coinduction, lambda calculus, standardization

Digital Object Identifier 10.4230/LIPIcs.TYPES.2011.16

1 Introduction

In the untyped lambda calculus [1], one observes that the fixed point combinator Y has Böhm tree

$$\lambda f.f(f(f\cdots))$$

which looks like a "limit" of the infinite reduction sequence

$$Y \to \lambda f.Yf \to \lambda f.f(Yf) \to \dots$$

Infinitary rewriting [6, 13, 2, 3, 17, 9] makes such statements precise by considering infinite reduction sequences together with the topology on infinite terms generated by finite prefixes: the basic opens are of the form

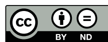
$$\mathcal{O}_{C[]} = \{t \mid \exists t_1,\dots,t_n.\, t = C[t_1,\dots,t_n]\}$$

where $C[]$ is a finite multi-hole context. Alternatively, this topology is given by the metric d where

$$d(s,t) = \inf\{2^{-n} \mid s \text{ and } t \text{ have the same symbols up to depth } n\}$$

Since the infinite terms can themselves be seen as formal limits of Cauchy sequences of finite terms with the metric above, it is natural to consider rewriting sequences together with this topological structure. Specifically, a reduction sequence

$$t_0 \to t_1 \to t_2 \to \cdots \to t_n \to \cdots$$


© J. Endrullis and A. Polonsky;
licensed under Creative Commons License BY-ND
18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson, Bengt Nordström; pp. 16–27

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

is said to *converge weakly to the limit* t if the sequence $\{t_i\}$ converges to t in the metric d.

For example, reducing Curry's fixed point combinator $Yf = WW$, where $W = \lambda x.f(xx)$, yields the infinite sequence

$$Yf = WW \to f(WW) \to f^2(WW) \to \cdots \to f^n(WW) \to \ldots$$

which converges to the limit f^ω.

However, the above notion of infinite reductions does not yet yield a satisfactory rewriting theory (intuitively, because topology does not respect the "rewriting structure" in any way). As has been often stressed by Jan Willem Klop, a much superior notion of transfinite reduction is the so-called *strongly convergent* reduction. This is a reduction as above which satisfies the additional condition that the depth of redexes contracted in the infinite sequence must tend to infinity. This constraint is sufficient to recover fundamental rewriting notions, including descendants, projections of reductions, and standardization.

In the present paper, we observe that an alternative, "coordinate-free" definition of strongly convergent reductions results from interpreting the binary reduction relation as a coinductive type family.

1.0.0.1 Related Work.

Catarina Coquand and Thierry Coquand have explored a similar approach in [4], giving a coinductive definition of standard reductions in infinitary combinatory logic. In his PhD thesis [11] and the paper [12], Felix Joachimski investigates finite reductions between coinductively defined infinite terms. To prove confluence, Joachimski introduces a coinductive definition of infinite developments, but not infinite reductions in general. Our proof of standardization for infinite reductions is a generalization of Plotkin's proof of standardization [15] for finitary rewriting; see also [19].

2 Setup

The set of infinite lambda terms is generated coinductively by the grammar

$$\Lambda^\infty ::= x \mid \Lambda^\infty\Lambda^\infty \mid \lambda x.\Lambda^\infty$$

For infinite terms s and t, we write $s = t$ if s and t are bisimilar, that is, the predicate $=$ is coinductively defined by:

$$\frac{}{x = x} \qquad\qquad \frac{s = s' \quad t = t'}{st = s't'} \qquad\qquad \frac{r = r'}{\lambda x.r = \lambda x.r'}$$

Thus $=$ is the largest relation R such that every $s\,R\,t$ is of one of the forms:

1. $x\;R\;x$,
2. $st\;R\;s't'$ for terms s, s', t, t' with $s\;R\;s'$ and $t\;R\;t'$, or
3. $\lambda x.r\;R\;\lambda x.r'$ for terms r, r' and a variable x with $r\;R\;r'$.

Here and henceforth, we use double inference lines to emphasize that the given derivation system defines a predicate or type family by coinduction rather than by induction.

We frequently denote regular infinite terms by systems of equations, e.g.:

$$M = (\lambda x.x)M$$

It is to be understood that the term M is the infinite tree unfolding of this equation, see Figure 1.

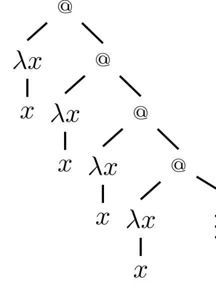


Figure 1 *A regular infinite term.*

The operation of capture-avoiding *substitution*, written $s[u/x]$, is defined by guarded corecursion

s	$s[u/x]$	
x	u	
y	y	$y \neq x$
$s_1 s_2$	$s_1[u/x]s_2[u/x]$	
$\lambda y.r$	$\lambda y.r[u/x]$	$y \notin \mathsf{FV}(u)$

We will not discuss here the problem of implementing Barendregt's variable convention in the infinitary setting. It does present an interesting issue: if the variables are represented by a countable set, then each variable might occur freely in a lambda term. Then it is not possible to find a fresh name which does not occur in it. (We note that the trick of Hilbert's Hotel is not applicable here, since we cannot rename free variables.)

In our Coq formalization, we have used classical deBruijn representation which successfully solves this problem, but it entails proving a number of lifting lemmas. Perhaps the most natural approach to formalizing infinitary rewriting would be to use an explicit substitution calculus based on explicit scope delimiters, as in [10], [18].

The substitution operator satisfies the following, provided that $x \notin \mathsf{FV}(u)$:

$$s[t/x][u/y] = s[u/y][t[u/y]/x] \tag{1}$$

The *one-step beta reduction* is a binary relation on Λ^∞, defined inductively by the rules

$$\frac{}{(\lambda x.r)t \longrightarrow r[t/x]} \qquad \frac{s \longrightarrow s'}{st \longrightarrow s't} \qquad \frac{t \longrightarrow t'}{st \longrightarrow st'} \qquad \frac{r \longrightarrow r'}{\lambda x.r \longrightarrow \lambda x.r'}$$

The relation $\twoheadrightarrow$ of a *finite* beta reduction is the reflexive-transitive closure of $\longrightarrow$, defined inductively by the rules

$$\frac{}{t \twoheadrightarrow t} \qquad \frac{s \twoheadrightarrow t \quad t \longrightarrow t'}{s \twoheadrightarrow t'}$$

(We note that we could also append the single beta step on the left.)

The notion of *one-step weak head reduction* $\longrightarrow_w$ is obtained by restricting $\longrightarrow$ to only the first two rules:

$$\frac{}{(\lambda x.r)t \longrightarrow_w r[t/x]} \qquad \frac{s \longrightarrow_w s'}{st \longrightarrow_w s't}$$

Correspondingly, $\twoheadrightarrow_w$ is the reflexive-transitive closure of $\longrightarrow_w$.

The *infinite* beta reduction $\twoheadrightarrow\!\!\!\!\rightarrow$ is defined coinductively by requiring that every node in the syntax tree becomes "frozen" after finitely many steps. This is made explicit by the following derivation rules, which are this time interpreted coinductively:

$$\frac{s \twoheadrightarrow x}{\overline{s \twoheadrightarrow\!\!\!\!\rightarrow x}}$$

$$\frac{s \twoheadrightarrow t_1 t_2 \qquad t_1 \twoheadrightarrow\!\!\!\!\rightarrow t_1' \qquad t_2 \twoheadrightarrow\!\!\!\!\rightarrow t_2'}{\overline{s \twoheadrightarrow\!\!\!\!\rightarrow t_1' t_2'}}$$

$$\frac{s \twoheadrightarrow \lambda x.r \qquad r \twoheadrightarrow\!\!\!\!\rightarrow r'}{\overline{s \twoheadrightarrow\!\!\!\!\rightarrow \lambda x.r'}}$$

▶ **Example 1.** Let us reconsider Curry's fixed point combinator $Y = \lambda f.WW$ with $W = \lambda x.f(xx)$. Then the infinite rewrite sequence $Yf \twoheadrightarrow\!\!\!\!\rightarrow f^\omega$ with $f^\omega = f(f^\omega)$ can be derived as follows:

Note that this is an infinite proof term, as indicated by the loop $\cdots\blacktriangleright$.

Classically, transfinite reduction sequences are defined as follows (here we view ordinals α as the set of all smaller ordinals $\alpha = \{\beta \mid \beta < \alpha\}$):

▶ **Definition 2.** Let $s \in \Lambda^\infty$, and let α be an ordinal.

A map $t : (\alpha \cup \{\alpha\}) \to \Lambda^\infty$, together with steps $\sigma_\beta : t(\beta) \to t(\beta+1))_{\beta<\alpha}$ for every $\beta < \alpha$, is a *strongly convergent reduction of length* α *from* $t(0)$ *to* $t(\alpha)$, if the following conditions hold:

1. If $\gamma \leq \alpha$ is a limit ordinal, then $t(\gamma)$ is the limit, in the metric topology on infinite terms, of the ordinal-indexed sequence $(t(\beta))_{\beta<\gamma}$;
2. If $\gamma \leq \alpha$ is a limit ordinal, then for every $d \in \mathbb{N}$, there exists $\beta < \gamma$, such that, for all β' with $\beta \leq \beta' < \gamma$, the redex contracted in the step $\sigma_{\beta'}$ occurs at depth greater than d.

The proof of the following theorem will be given in Section 4.

▶ **Theorem 3.** *$s \twoheadrightarrow\!\!\!\!\rightarrow t$ if and only if s reduces to t via a strongly convergent reduction sequence.*

One advantage of the coinductive approach is that it provides a simple and natural definition of standard reductions.

The *infinitary standard reduction* is obtained by the same rules as the infinite beta reductions, except that the finite prefixes are now required to be weak head reductions.

$$\frac{s \twoheadrightarrow_w x}{\overline{s \twoheadrightarrow\!\!\!\!\rightarrow_s x}}$$

$$\frac{s \twoheadrightarrow_w t_1 t_2 \qquad t_1 \twoheadrightarrow\!\!\!\!\rightarrow_s t_1' \qquad t_2 \twoheadrightarrow\!\!\!\!\rightarrow_s t_2'}{\overline{s \twoheadrightarrow\!\!\!\!\rightarrow_s t_1' t_2'}}$$

$$\frac{s \twoheadrightarrow_w \lambda x.r \qquad r \twoheadrightarrow\!\!\!\!\rightarrow_s r'}{\overline{s \twoheadrightarrow\!\!\!\!\rightarrow_s \lambda x.r'}}$$

3 Standardization

We now seek to prove the following fact:

$$s \longrightarrow\!\!\!\!\!\ggg t \quad\Longrightarrow\quad s \longrightarrow\!\!\!\!\!\ggg_s t$$

The intuition is as follows. In order to replace beta-prefixes with weak head-prefixes, we standardize the beta prefix, extract the initial weak head reduction, and absorb the remainder into the coinductive call. However, the standardization of a finite beta reduction can give rise to an infinite reduction, as in the following counterexample to the Church–Rosser theorem for *finite* reductions between infinite terms:

$$(\lambda f.f^\omega)(\mathtt{I}x) \longrightarrow (\lambda f.f^\omega)x \longrightarrow x^\omega$$

when standardized, yields

$$(\lambda f.f^\omega)(\mathtt{I}x) \longrightarrow (\mathtt{I}x)^\omega \longrightarrow\!\!\!\!\!\ggg x^\omega$$

As an intermediate step, we therefore first convert the prefixes to infinite standard reductions. This suggests the introduction of one more auxiliary reduction $\longrightarrow\!\!\!\!\!\ggg_a$, which follows the above scheme but takes for prefixes infinite standard reductions defined previously.

$$\begin{array}{c} s \longrightarrow\!\!\!\!\!\ggg_s x \\ \hline\hline s \longrightarrow\!\!\!\!\!\ggg_a x \end{array}$$

$$\begin{array}{c} s \longrightarrow\!\!\!\!\!\ggg_s t_1t_2 \qquad t_1 \longrightarrow\!\!\!\!\!\ggg_a t_1' \qquad t_2 \longrightarrow\!\!\!\!\!\ggg_a t_2' \\ \hline\hline s \longrightarrow\!\!\!\!\!\ggg_a t_1't_2' \end{array}$$

$$\begin{array}{c} s \longrightarrow\!\!\!\!\!\ggg_s \lambda x.r \qquad r \longrightarrow\!\!\!\!\!\ggg_a r' \\ \hline\hline s \longrightarrow\!\!\!\!\!\ggg_a \lambda x.r' \end{array}$$

Infinitary standardization theorem now follows by a series of simple lemmas:

▶ **Lemma 4.** *We have*

1. $s \twoheadrightarrow_w t,\ t \twoheadrightarrow_w u \Longrightarrow s \twoheadrightarrow_w u$
2. $s \twoheadrightarrow_w t,\ t \longrightarrow\!\!\!\!\!\ggg_s u \Longrightarrow s \longrightarrow\!\!\!\!\!\ggg_s u$
3. $s \longrightarrow\!\!\!\!\!\ggg_s s',\ t \longrightarrow\!\!\!\!\!\ggg_s t' \Longrightarrow s[t/x] \longrightarrow\!\!\!\!\!\ggg_s s'[t'/x]$
4. *For* $\longrightarrow_R \in \{\longrightarrow, \twoheadrightarrow, \twoheadrightarrow_w\}$,

   $$s \longrightarrow\!\!\!\!\!\ggg_s t, t \longrightarrow_R u \Longrightarrow s \longrightarrow\!\!\!\!\!\ggg_s u$$

5. $s \longrightarrow\!\!\!\!\!\ggg_s t,\ t \longrightarrow\!\!\!\!\!\ggg_s u \Longrightarrow s \longrightarrow\!\!\!\!\!\ggg_s u$

Proof. **1.** By induction.

2. By case distinction, using 1 to concatenate the prefix.

3. By coinduction, using that

$$\begin{array}{lcl} s \longrightarrow_w t & \Longrightarrow & s[u/x] \longrightarrow_w t[u/x] \\ s \twoheadrightarrow_w t & \Longrightarrow & s[u/x] \twoheadrightarrow_w t[u/x] \end{array}$$

4. By induction on $t \longrightarrow_R u$, using 3 for the redex base case.

5. By coinduction on $t \longrightarrow\!\!\!\!\!\ggg_s u$

Case 1 $t \twoheadrightarrow_w x = u$. Then $s \longrightarrow\!\!\!\!\!\ggg_s x$ by 4.

Case 2 $u = u_1u_2$, $t \twoheadrightarrow_w t_1t_2$, and $t_i \longrightarrow\!\!\!\!\!\ggg_s u_i$. By 4, $s \longrightarrow\!\!\!\!\!\ggg_s t_1t_2$. Hence $s \twoheadrightarrow_w t_1't_2'$, with $t_i' \longrightarrow\!\!\!\!\!\ggg_s t_i$. By coinduction, $t_i' \longrightarrow\!\!\!\!\!\ggg_s u_i$. Using that $s \twoheadrightarrow_w t_1't_2'$, we get $s \longrightarrow\!\!\!\!\!\ggg_s u_1u_2$.

Case 3 $u = \lambda x.v$, $t \twoheadrightarrow_w \lambda x.r$, and $r \twoheadrightarrow_s v$. By 4, $s \twoheadrightarrow_s \lambda x.r$. Hence $s \twoheadrightarrow_w \lambda x.r'$, with $r' \twoheadrightarrow_s r$. By coinduction, $r' \twoheadrightarrow_s v$. Using that $s \twoheadrightarrow_w \lambda x.r'$, we get $s \twoheadrightarrow_s \lambda x.v$. ◀

▶ **Lemma 5.** *We have*

1. $s \twoheadrightarrow_s t,\ t \twoheadrightarrow_s u \Longrightarrow s \twoheadrightarrow_s u$
2. $s \twoheadrightarrow_s t,\ t \twoheadrightarrow_a u \Longrightarrow s \twoheadrightarrow_a u$.
3. $s \twoheadrightarrow_a s',\ t \twoheadrightarrow_a t' \Longrightarrow s[t/x] \twoheadrightarrow_a s'[t'/x]$
4. *For* $\longrightarrow_R \in \{\longrightarrow, \twoheadrightarrow, \twoheadrightarrow_w, \twoheadrightarrow_s\}$,

$$s \twoheadrightarrow_a t, t \longrightarrow_R u \Longrightarrow s \twoheadrightarrow_a u$$

5. $s \twoheadrightarrow_a t,\ t \twoheadrightarrow_a u \Longrightarrow s \twoheadrightarrow_a u$

Proof. 1 was proved in the previous lemma. The rest follows the proof there mutatis mutandis. ◀

▶ **Lemma 6.** *We have*

1. $s \twoheadrightarrow_s t \Longrightarrow s \twoheadrightarrow t$
2. $s \twoheadrightarrow t \Longrightarrow s \twoheadrightarrow_s t$
3. $s \twoheadrightarrow t \Longrightarrow s \twoheadrightarrow_a t$
4. $s \twoheadrightarrow_s t \Longrightarrow s \twoheadrightarrow_a t$
5. $s \twoheadrightarrow_a t \Longrightarrow s \twoheadrightarrow_s t$

Proof. **1.** Immediate: every weak head prefix is also a beta prefix.

2. By induction on $s \twoheadrightarrow t$, using Lemma 4.4 and reflexively of $\twoheadrightarrow_s$.

3. Immediate by 2.

4. By composition of 1 and 3.

5. By coinduction on $s \twoheadrightarrow_a t$:

Case 1 $s \twoheadrightarrow_s x = t$. Done.

Case 2 $t = t_1 t_2$, $s \twoheadrightarrow_s s_1 s_2$, and $s_i \twoheadrightarrow_a t_i$. Hence $s \twoheadrightarrow_w s'_1 s'_2$, with $s'_i \twoheadrightarrow_s s_i$. By 4, $s'_i \twoheadrightarrow_a s_i$. By Lemma 5.5, $s'_i \twoheadrightarrow_a t_i$. By coinduction, $s'_i \twoheadrightarrow_s t_i$. Using that $s \twoheadrightarrow_w s'_1 s'_2$, we get $s \twoheadrightarrow_s t_1 t_2$ by constructor.

Case 3 $t = \lambda x.v$, $s \twoheadrightarrow_s \lambda x.r$, and $r \twoheadrightarrow_a v$. Hence $s \twoheadrightarrow_w \lambda x.r'$, with $r' \twoheadrightarrow_s r$. By 4, $r' \twoheadrightarrow_a r$. By Lemma 5.5, $r' \twoheadrightarrow_a v$. By coinduction, $r' \twoheadrightarrow_s v$. Using that $s \twoheadrightarrow_w \lambda x.r'$, we get $s \twoheadrightarrow_s \lambda x.v$.

◀

▶ **Theorem 7.** $s \twoheadrightarrow t \quad \Longrightarrow \quad s \twoheadrightarrow_s t$

Proof. By composing parts 3 and 5 of Lemma 6. ◀

▶ Remark. Technically speaking, we have only proved the existence part of Curry's standardization theorem; as some rewriting theorists would argue, in the finitary case, the theorem also asserts that the standard reduction is strongly equivalent with the given one in the sense of Lévy, and is furthermore a unique representative of this equivalence class.

We find it an interesting problem to give a coinductive formulation of the notion of Lévy-equivalence for infinite reductions.

The Coq formalization of the coinductive treatment of infinitary rewriting — in particular, the proof of standardization — can be downloaded from `http://joerg.endrullis.de`. All coinductive proofs in Coq have to adhere to a strict syntactic guardedness condition [5] for guaranteeing constructive well-definedness, also known as productivity [7]. We have employed a proof transformation method from [8], in order to transform productive into guarded proofs.

4 Coinductive Reductions are Strongly Convergent

We now prove Theorem 3:

$$s \twoheadrightarrow t \iff s \text{ reduces to } t \text{ via a strongly convergent reduction sequence}$$

Theorem 3. ($\Rightarrow$) Suppose that $s \twoheadrightarrow t$. By traversing the infinite derivation tree of $s \twoheadrightarrow t$ in the breadth-first order, and accumulating the finite beta-prefixes by concatenation, we get a reduction sequence of length ω which satisfies the depth requirement by construction.

($\Leftarrow$) Let R be a strongly convergent reduction sequence from s to t of length α; we write this as $s \xrightarrow{R}_\alpha t$. By induction on α, we show that $s \twoheadrightarrow_a t$. This suffices for $s \twoheadrightarrow t$ by Lemma 6.5 and 6.1.

Zero case: $s \xrightarrow{R}_0 t$. Then $s = t$, hence $s \twoheadrightarrow_s t$ and $s \twoheadrightarrow_a t$.

Successor: $s \xrightarrow{R}_{\alpha+1} t$. Then $s \xrightarrow{R}_\alpha s' \longrightarrow t$. Then $s' \twoheadrightarrow_s t$ and $s' \twoheadrightarrow_a t$, and by the induction hypothesis, $s \twoheadrightarrow_a s'$. Thus $s \twoheadrightarrow_a t$ by Lemma 5.5.

Limit: $s \xrightarrow{R}_\alpha t$, α a limit ordinal. We define an infinite derivation of $s \twoheadrightarrow t$ coinductively. By the depth condition, there exists $\beta < \alpha$ such that, for every $\gamma \geq \beta$, the redex contracted by R at γ occurs at depth greater than zero. Let t_β be the term at index β in R. Then by induction hypothesis we have $s \twoheadrightarrow_a t_\beta$, and $s \twoheadrightarrow_s t_\beta$ by Lemma 6.5. We distinguish three possible shapes of t_β.

- **Variable:** $t_\beta = x$. This is impossible, since then t_β cannot reduce to anything, while we assumed that $\beta < \gamma$.
- **Abstraction:** $t_\beta = \lambda x.r$. Then $t = \lambda x.u$, and $r \longrightarrow_{\leq\alpha} u$. Then $r \twoheadrightarrow_a u$ by coinduction. Now $s \twoheadrightarrow \lambda x.u$ by the abstraction constructor of $\twoheadrightarrow_a$.
- **Application:** $t_\beta = t_1 t_2$. Then $t = u_1 u_2$ and the tail of reduction R past β can be split into two parts $\{t_i \longrightarrow_{\leq\alpha} u_i \mid i = 0, 1\}$ of length at most α. Then $t_0 \twoheadrightarrow_a u_0$ and $t_1 \twoheadrightarrow_a u_1$ by coinduction. Now $s \twoheadrightarrow u_1 u_2$ by the application constructor of $\twoheadrightarrow_a$.

◀

5 Loops Loops Loops Loops Loops Loops Loops Loops Loops ...

One might wonder which infinite reductions converge in the weak sense of topology but not in the strong/coinductive sense above. One example is the infinite head reduction of $\Omega = (\lambda x.xx)(\lambda x.xx)$.

$$\Omega \to \Omega \to \Omega \to \cdots \tag{2}$$

which converges to Ω in the metric on infinite terms, but is not strongly convergent. Here we nevertheless have $\Omega \twoheadrightarrow \Omega$ due to finite prefixes of the infinite reduction (in particular, the empty reduction). Not every topologically convergent reduction has a strongly convergent counterpart. This is illustrated by the following reduction:

$$\begin{aligned} M &= (\lambda x_0.(\lambda x_1.(\lambda x_2.\ldots)(x_1 \mathsf{I}))(x_0 \mathsf{I}))\mathsf{I} \\ &\to (\lambda x_0.(\lambda x_1.(\lambda x_2.\ldots)(x_1 \mathsf{I}))(x_0 \mathsf{I}))(\mathsf{II}) \\ &\to (\lambda x_0.(\lambda x_1.(\lambda x_2.\ldots)(x_1 \mathsf{I}))(x_0 \mathsf{I}))(\mathsf{III}) \\ &\quad \vdots \\ &\to (\lambda x_0.(\lambda x_1.(\lambda x_2.\ldots)(x_1 \mathsf{I}))(x_0 \mathsf{I}))(\mathsf{I}^\omega) = N \end{aligned} \tag{3}$$

This reduction converges only topologically, every rewrite step occurs at the root. In fact, there exists no strongly convergent reduction from M to N, we do not have $M \twoheadrightarrow N$.

We note that both examples of topologically convergent reductions (2) and (3) contain a term that admits a loop: $\Omega \to \Omega$ and $N \to N$, respectively. A recent theorem of [16] states that these examples are paradigmatic: if R is a reduction sequence which is weakly, but not strongly, convergent, then R contains a term which reduces to itself in one beta-reduction step.

We conclude this paper by giving a characterization of all such terms.

▶ **Definition 8.** For $M \in \Lambda^\infty$, we define:

1. A *one-cycle* is a rewrite step $M \to M$.
2. A *loop* is a rewrite step $M \to M$ at the root of the term.

Note that every one-cycle $M \to M$ is of the form $M \equiv C[M'] \to C[M']$ for some context C and a loop $M' \to M'$. As a consequence, the interesting objects are the loops, and we are interested in a characterization of terms that admit loops. For the case of (ordinary) finitary λ-calculus, this problem has been studied and solved by Lercher in 1976 [14] who showed that Ω is the only finite looping λ-term:

▶ **Theorem 9** (Lercher). *The only finite λ-term M such that $M \to M$ via a root step is $\Omega \equiv (\lambda x.\, xx)(\lambda x.\, xx)$.*

In infinitary lambda calculus, the situation becomes more involved. It turns out, that there are 3 looping terms with a finite spine (among which of course Ω), and there is a whole scheme of uncountably many terms with an infinite spine.

▶ **Theorem 10.** *The looping terms in infinitary λ-calculus are precisely the terms that are of one of the following forms:*

1. I^ω,
2. $\Omega \equiv (\lambda x.\, xx)(\lambda x.\, xx)$,
3. *BB where B is the infinite solution of $B \equiv \lambda x.xB$, or*
4. *$(\lambda x_0.(\lambda x_1.(\lambda x_2....)s_2)s_1)s_0$ such that for every $i \in \mathbb{N}$, the term s_{i+1} is obtained from s_i by replacing all x_j by x_{j+1} followed by replacing an arbitrary (possibly infinite) number of occurrences of s_0 by x_0. We call such a term a* cascade.

The terms in cases (1), (2) and (3) are displayed in Figure 2.

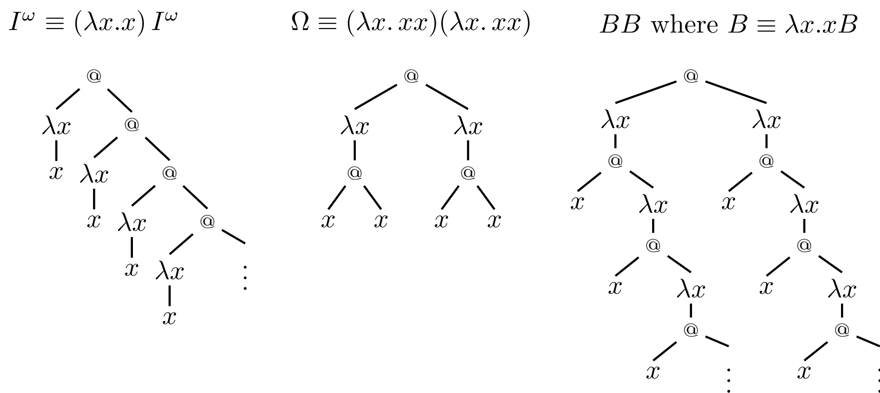
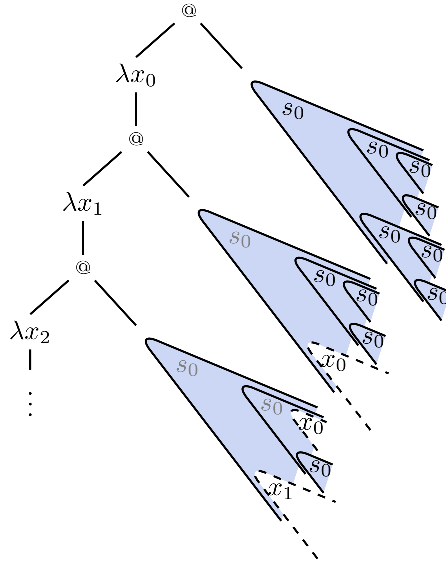


Figure 2 *Looping terms in infinitary λ-calculus, except for cascades.*

The case (4) of cascades is illustrated in Figure 3, and an example of a cascade is shown in Figure 4. A cascade $(\lambda x_0.(\lambda x_1.(\lambda x_2....)s_2)s_1)s_0$ can equivalently be characterized as follows: for every $n \in \mathbb{N}$, the term s_i is obtained from s_{i+1} by a substitution replacing x_0 by s_0 and all variables x_{j+1} by x_j.

4th (class of) solution(s): $M \equiv (\lambda x_0.(\lambda x_1.(\lambda x_2.\ldots)s_2)s_1)s_0$
with $s_i = s_{i+1}[\, x_0 = s_0,\ \ x_1 = x_0, \ldots, x_{i+1} = x_i \,]$ for $i \geq 1$



The recipe for cascades:

- take any term s_0
- obtain s_{i+1} from s_i by:
  (a) replacing all occurrences of x_i by x_{i+1} (for all $i \in \mathbb{N}$ in parallel),
  (b) replacing some (zero or more) occurrences of subterms s_0 by x_0

Figure 3 *The structure of cascades in infinitary λ-calculus. The gray occurrences s_0 indicate that this term is obtained from s_0 by replacing subterms by variables.*

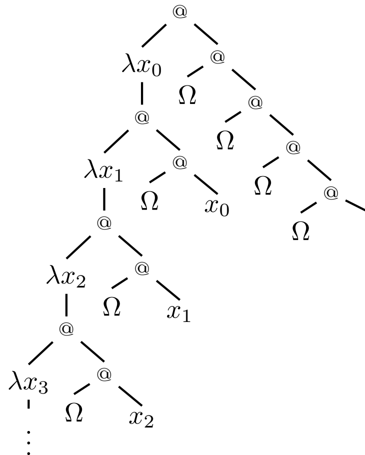


Figure 4 Example of a cascade.

Proof of Theorem 10. Let $M \in \Lambda^\infty$ be a term that admits a loop $M \to M$. Then M has a redex at the root, thus $M \equiv (\lambda x.M')C$ for some $M', C \in \Lambda^\infty$. We distinguish the following cases for M':

(ia) *M' is a variable, $M' \equiv x$.* Then $M \equiv (\lambda x.x)C \to C \equiv M$, and hence $M \equiv \mathsf{I}^\omega$. This is case (1) in the theorem.

(ia) *M' is a variable, $M' \equiv y \neq x$.* Then $M \equiv (\lambda x.y)C \to y \not\equiv M$, contradiction.

(ii) *M' is an abstraction.* Then the reduct would be an abstraction, contradiction

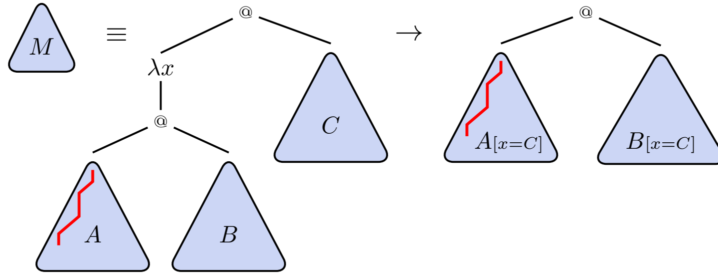
(iii) *M' is an application, $M \equiv AB$.* We analyse this case below.

For (iii) we have: $M \equiv (\lambda x.AB)C$ and by assumption $M \equiv (AB)[x := C]$. Hence

(a) $A[x := C] \equiv \lambda x.AB$, and

(b) $C \equiv B[x := C]$.

We consider the left spine L of A, depicted thick and red in the following picture:

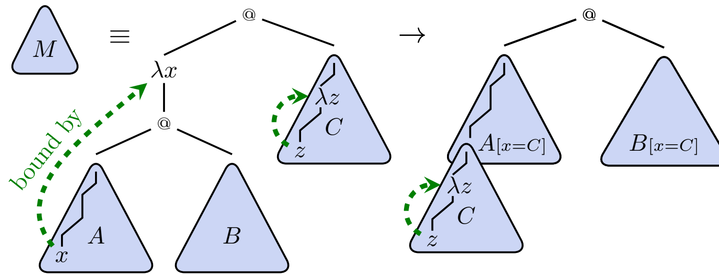


Now there are two possibilities, either the spine L is finite or infinite:

(1) *L is finite.*

Assume that the spine would end in a variable $y \not\equiv x$. This assumption yields a contradiction by (a) since then the spine of $A[x := C]$ in the reduct would be shorter than the left spine of $(\lambda x.AB)$.

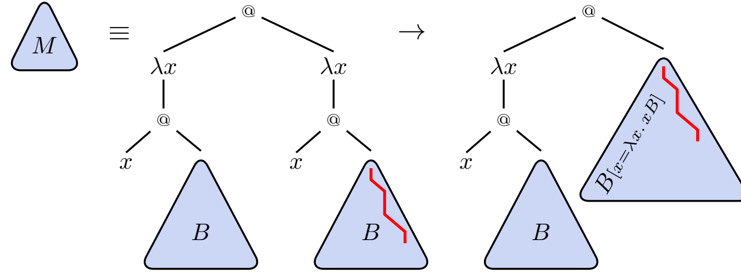
As a consequence, the spine ends in the variable x. This situation is surveyed in the following picture:



We conclude that $A \equiv x$ as otherwise the variable at the end of the spine in $A[x := C]$ cannot be bound at the root as in $(\lambda x.AB)$. Then $C \equiv \lambda x.xB$ by (a) and together with (b) we get:

$$\lambda x.xB \equiv B[x := \lambda x.xB] \qquad (\dagger)$$

We consider the right spine R of B, displayed red in the following picture:



Again, there are the following possibilities:

(i) *R is finite.* As before, it follows that $B \equiv x$ since otherwise the right spine of the reduct would be shorter than the right spine of M. Hence we have found the well-known looping term $M \equiv \Omega \equiv (\lambda.xx)(\lambda.xx)$.

(ii) *R is infinite.* Then the right spine of $\lambda x.xB$ is the same as that of B, and hence is an alternation of abstraction and application. Thus:

$$B \equiv \lambda x_0.s_0(\lambda x_1.s_1(\lambda x_2.s_2(\ldots)))$$

for some terms s_i. From (†) it follows $s_0 \equiv x_0$, and this in turn implies that $s_1 \equiv x_1$, and then $s_2 \equiv x_2$, ans so forth. Using induction we obtain $s_i \equiv x_i$. Thus $B \equiv \lambda x.xB$, $C \equiv B$ and $M \equiv (\lambda x.xB)B \equiv BB$.

(2) *L is infinite.*

Then the spine of A must be the same as that of $(\lambda x.AB)$, and thus is an alteration of lambda and application. As a consequence, we have

$$M \equiv (\lambda x_0.(\lambda x_1.(\lambda x_2....)s_2)s_1)s_0$$

for some terms s_i. As a consequence the loop $M \to M$, it follows that:

$$M \equiv (\lambda x_0.(\lambda x_1.(\lambda x_2....)s_2)s_1)s_0 \;=_\alpha\; (\lambda x_1.(\lambda x_2....)s_2)s_1)[x_0 := s_0]$$

Thus, for every $i \geq 1$ we have that s_i is be obtained from s_{i+1} by replacing x_0 by s_0 and all variables x_{j+1} by x_j (the α-renaming).

◀

References

1. H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier Science, revised edition, 1985.
2. A. Berarducci. Infinite λ-Calculus and Non-Sensible Models. In *Logic and Algebra (Pontignano, 1994)*, pages 339–377. Dekker, New York, 1996.
3. A. Berarducci and B. Intrigila. Church–Rosser λ-theories, Infinite λ-calculus and Consistency Problems. *Logic: From Foundations to Applications*, pages 33–58, 1996.
4. C. Coquand and T. Coquand. On the Definition of Reduction for Infinite Terms. *Comptes Rendus de l'Académie des Sciences. Série I*, 323(5):553–558, 1996.
5. Th. Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24–28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1994.

6 N. Dershowitz, S. Kaplan, and D.A. Plaisted. Rewrite, Rewrite, Rewrite, Rewrite, Rewrite,.... *Theoretical Computer Science*, 83(1):71–96, 1991.
7 J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J.W. Klop. Productivity of Stream Definitions. *Theoretical Computer Science*, 411:765–782, 2010.
8 J. Endrullis, D. Hendriks, and M. Bodin. Circular Coinduction in Coq using Bisimulation-Up-To Techniques. Unpublished note.
9 J. Endrullis, D. Hendriks, and J.W. Klop. Highlights in Infinitary Rewriting and Lambda Calculus. *Theoretical Computer Science*, 464:48–71, 2012.
10 D. Hendriks and V. van Oostrom. Adbmal. In *Proc. Conf. on Automated Deduction (CADE 2003)*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 136–150. Springer, 2003.
11 F. Joachimski. *Reduction Properties of* Π*IE-Systems*. PhD thesis, LMU München, 2001.
12 F. Joachimski. Confluence of the Coinductive [Lambda]-Calculus. *Theoretical Computer Science*, 311(1-3):105–119, 2004.
13 J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995.
14 B. Lercher. Lambda-Calculus Terms That Reduce To Themselves. *Notre Dame Journal of Formal Logic*, 17(2):291–292, 1976.
15 G.D. Plotkin. Call-by-Name, Call-by-Value and the Lambda-Calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.
16 J.G. Simonsen. Weak Convergence and Uniform Normalization in Infinitary Rewriting. In *Proc. 20th Int. Conf. on Rewriting Techniques and Applications (RTA 2009)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 311–324. Schloss Dagstuhl, 2010.
17 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
18 V. van Oostrom. Explicit Substitution for Graphs. In *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, number 9, pages 34–39, 2005.
19 H. Xi. Upper bounds for standardizations and an application. *J. Symb. Log.*, 64(1):291–303, 1999.

A new approach to the semantics of model diagrams

Johan G. Granström

Google
Brandschenkestrasse 110
8002 Zürich, Switzerland
georg.granstrom@acm.org

Department of Computer Science
King's College London
Strand, London, WC2R 2LS, U.K.

Abstract

Sometimes, a diagram can say more than a thousand lines of code. But, sadly, most of the time, software engineers give up on diagrams after the design phase, and all real work is done in code. The supremacy of code over diagrams would be leveled if diagrams were code. This paper suggests that model and instance diagrams, or, which amounts to the same, class and object diagrams, become first level entities in a suitably expressive programming language, viz., type theory.

The proposed semantics of diagrams is compositional and self-describing, i.e., reflexive, or metacircular. Moreover, it is well suited for metamodelling and model driven engineering, as it is possible to prove model transformations correct in type theory. The encoding into type theory has the additional benefit of making diagrams immediately useful, given an implementation of type theory.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, F.3.2 Semantics of Programming Languages.

Keywords and phrases model diagram, modelling, metamodelling, semantics, compositionality, self-description, metacircularity, reflexivity, universal model, MOF, UML.

Digital Object Identifier 10.4230/LIPIcs.TYPES.2011.28

1 Introduction

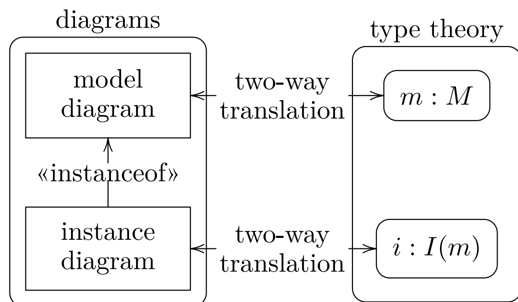


Figure 1 A modelling language (M, I) with corresponding model and instance diagrams.

The semantics of visual modelling languages, such as UML class diagrams, is surrounded by much confusion [21]. On the other hand, much is gained from using diagrams, as the same diagram can be understood to different degrees and from different angles by collaborators. In addition, with today's rapidly changing codebases, any documentation external to code is doomed to soon be out of date [30]. Consequently, documentation, in the form of diagrams, that is guaranteed to be in synch with code, because it is code, is worth much more than mere documentation.

© Johan G. Granström;
licensed under Creative Commons License BY-ND
18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson, Bengt Nordström; pp. 28–40
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Model diagrams can be translated to linear notation (Figure 1 and Sect. 4), and this linear notation can be completely formalized (Sect. 5) in a suitably expressive programming language like intuitionistic type theory [24, 16] or the calculus of constructions [8]. A benefit of translating into an expressive language is that model transformations can be proved correct [29, 14]. In addition, a direct translation into an executable language, such as type theory, has the pragmatic value of making models immediately useful when programming.

One important property of the suggested translation, from diagrams to linear notation, is that the resulting semantics is compositional. That is, a small addition to the diagram cannot give rise to a large change in its meaning. For example, the notion of inheritance is difficult to understand compositionally, as adding an inheritance relation between two classes (a small addition) may create an inheritance cycle (a large change in meaning). This phenomenon is further discussed in Sect. 8, and the modelling language of Figure 9 uses generalisation instead of inheritance to preserve compositionality.

The translation from diagrams to type theory will first be applied to a simple modelling language (Figure 4) with only three notions, and then to a less simple language (Figure 9). Both of these modelling languages are self-describing (Sect. 2 and the Theorem). That is, there is a particular model of the language, that describes the whole language.

Turing's discovery [34] of the universal machine, capable of interpreting any program, was of paramount importance as it lead to the design of the stored program computer [11]. The dichotomy between code and data makes it plausible that analogues of Turing's universal machine in the space of data, i.e., self-describing modelling languages, are more important than currently appreciated. This is one reason for studying self-describing modelling languages: further motivation is given in Sect. 2.

	an element of M is	an element of $I(m)$ is
UML	a class diagram	an instance of m
MOF	a metamodel	a metamodel instance of m
DSD	a DSD schema	a document valid w.r.t. m
EBNF	an EBNF grammar	a string conforming to m
RDB	a database schema	a database instance of m
types	a type	an object of type m

Table 1 Examples of modelling languages of different kinds: syntax description languages, like EBNF [35], XML schema languages, like DSD [26], the language of relational databases (RDB) [7], and any type system, fit the definition of modelling language.

2 Self-describing modelling languages

A pair

$$\begin{cases} M & : & \text{set} \\ I & : & M \to \text{set}. \end{cases} \tag{1}$$

will be called a *modelling language*.[1] In a given modelling language (M, I), an element of M is called a *model*, and an element of $I(m)$, for a model m, is called an *instance* of m. An example of a modelling language is displayed in Figure 2. It has two models, and each model has two instances. There are many interesting examples of modelling languages according to this definition, not all of them with a corresponding visual notation. Some noteworthy examples are given in Table 1.

[1] Or, to be more precise, a *formal modelling language*. This structure is known elsewhere in the literature as *world* [19, 17] or *container* [22].

A *universal model* of a modelling language (M, I) is a model $u : M$ where the set $I(u)$ is isomorphic to M. The parts of the isomorphism will be named ρ (reflection) and π (reification), i.e., the diagram

$$I(u) \underset{\pi}{\overset{\rho}{\rightleftarrows}} M \tag{2}$$

commutes. In particular, $\pi(u) : I(u)$. A modelling language will be called *self-describing*, *metacircular*, or *reflexive*, if it has a universal model.[2]

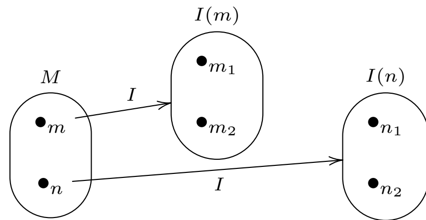


Figure 2 The leftmost oval shape represents the set of all models M in a modelling language (M, I).

For example, the DSD schema language for XML is self-describing in the sense that an XML document is a well-formed DSD schema if and only if it validates against the universal DSD schema [26, § 4]. Other schema languages for XML lack this feature.

Wirth succinctly describes the gist of EBNF's syntax by a universal EBNF grammar (Table 2). The only notions that remain to be explained are *character* and *identifier*. See Wirth's communication [35] for details. EBNF is probably the most concise self-describing language in current use.

There are at least three reasons why a modelling language should admit a (natural) universal model.

syntax	= { production }.
production	= identifier "=" expression ".".
expression	= term { "|" term}.
term	= factor { factor }.
factor	= identifier | literal | "(" expression ")" | "[" expression "]" | "{" expression "}".
literal	= """" character { character } """".

Table 2 The syntax of EBNF described by a EBNF grammar, verbatim after Wirth [35].

(1) The same query language can be used to query user models and metamodels alike. Relational database administrators have used this feature for decades to query the information schema [23]. Strictly speaking, only reification (π) is required for this to work. But at least a partial inverse ρ of π is needed if the results are to be useful.

(2) A modelling language that is not self-describing lacks, in a sense, expressivity, viz., the features necessary to describe itself. Moreover, a universal model exhibits a consistency among the notions used to explain the modelling language, and works as a kind of sanity check. The discussion about the notion of identifier in Sect. 7 exemplifies this form of sanity checking.

(3) The four layers of the OMG[3] pyramid [33] can be reduced to three, viz., the level of real-world entities (M0), the level of model instances (M1), and the level of models (M2). Given a modelling language (M, I), elements of M are M2 models and elements of $I(m)$, for an M2 model m, are M1 models. Clearly, a universal model $u : M$ resides in the M2 layer, despite being, as it were, a metamodel.

---

[2] To be precise, we should say that a model $u : M$ of a modelling language (M, I), is *universal with respect to an isomorphism* (ρ, π) *between* $I(u)$ *and* M. If the isomorphism is, as it were, unnatural, so is the universality of u.

[3] OMG (Object Management Group) is an international not-for-profit computer industry consortium and standards organization, responsible for, among other things, UML (Universal Modelling Language) and MOF (Meta-Object Facility).

3 A simple type system

Another example of a self-describing modelling language is the type system (D, T), that will be used in the definition of the simple modelling language (Sect. 5). It is defined by

$$\mathrm{D} = \{\mathrm{string}, \mathrm{money}, \mathrm{type}\}, \tag{3}$$

and

$$\begin{aligned} \mathrm{T}(\mathrm{string}) &= \{\textit{character strings}\} \\ \mathrm{T}(\mathrm{money}) &= \{\textit{monetary amounts}\} \\ \mathrm{T}(\mathrm{type}) &= \{\mathrm{string}, \mathrm{money}, \mathrm{type}\}. \end{aligned} \tag{4}$$

In particular, $\mathrm{T}(\mathrm{type}) = \mathrm{D}$, so 'type' is a universal model with ρ and π the identity function.

This rudimentary type system can be extended in several directions. For example, any number of basic types can be added, and the set D can be made closed under sum, product, and function space.

However, there are limitations on how the set of datatypes can be extended while maintaining the rule that type : T(type). It is for example known that the addition of the rule $\mathrm{U} : \mathrm{T}(\mathrm{U})$ to the rules for the type-theoretic universe U [24] leads to the paradox discovered by Girard [15].

4 From model diagrams to telescopes

Data modelling is first and foremost a *process*: relational modelling [7], entity-relationship modelling [6], object-role modelling [18], model driven engineering [30], etc. This process typically results in a set of diagrams. However, we are not trying to formalize the modelling process or the resulting diagrams, but the meanings underlying the diagrams. This is nontrivial, as, what a diagram *refers to*, *denotes*, or *means*, is elusive.

A first attempt is to say that a diagram refers to a *state of affairs*, so that, e.g., the symbol *Employee* of Figure 3 refers to a set of employees, etc. The problem with this explanation is that the diagram's state of affairs typically changes over time, so the diagram does not refer to any *particular* state of affairs: rather, the diagram signifies something general that various states of affairs fall under. That is, the entities of a diagram are *variable*, just as the relations of relational databases [10, pp. 17–18], [7, p. 4].

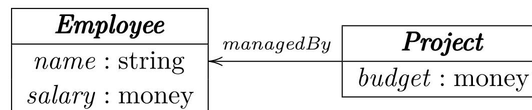


Figure 3 The model EP in the simple modelling language consists of two classes with attributes and a function between them.

The next observation is that, if there is to be any hope of systematically assigning meanings to diagrams, the meaning of a diagram somehow has to be composed of the meanings of its constituent parts. That is, the language behind the diagram has to adhere to the *principle of compositionality*, familiar from the philosophy of language [16, pp. 6–8]. Put differently, the meaning of a diagram should not change much due to a small change in the diagram.

To simplify the interpretation of diagrams, the following conventions will be adopted.

(1) A slanted font is used for uninterpreted symbols (e.g., *Employee*) and an upright font for interpreted symbols (e.g., string).

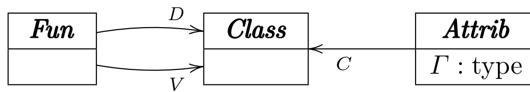


Figure 4 The universal model U of the simple modelling language: note that each construct of the modelling language (class, attribute, and function) is used by U.

(2) Interpreted symbols (e.g., money) may occur any number of times, whereas, if an uninterpreted symbol occurs more than once, it must be possible to disambiguate it.

(3) Uninterpreted symbols of a diagram range over certain categories of a formal language (e.g., *salary* ranges over money and *Project* ranges over the category of classes).

These conventions are best explained by taking Figure 3 as an example. Imagine a simple modelling language with only three notions: *class*, *attribute of class*, and *function between classes*.

In this language, Figure 3 is completely described by the following six assertions:

(1) *Project* is a class.

(2) *Employee* is a class.

(3) *budget* is an attribute of *Project* of type money.

(4) *name* is an attribute of *Employee* of type string.

(5) *salary* is an attribute of *Employee* of type money.

(6) *managedBy* is a function from *Project* to *Employee*.

The same assertions can be succinctly expressed using a yet to be defined formal language:

$$\begin{cases} \mathit{Project} : \text{class} \\ \mathit{Employee} : \text{class} \\ \mathit{budget} : \text{attrib}(\mathit{Project}, \text{money}) \\ \mathit{name} : \text{attrib}(\mathit{Employee}, \text{string}) \\ \mathit{salary} : \text{attrib}(\mathit{Employee}, \text{money}) \\ \mathit{managedBy} : \text{fun}(\mathit{Project}, \mathit{Employee}) \end{cases} \tag{5}$$

Such a sequence of assertions is similar to what a mathematician would write on the black board at the outset of an investigation: much like setting the stage for a play.

Now, we take a step back and recognize the above as a sequence of variable declarations. Thus, we have arrived at what de Bruijn [12] called a *telescope* and completed the informal path from model diagrams to telescopes. The reader is not required to be familiar with de Bruijn's telescopes, as the notion will only be used for purposes of comparison.

5 A simple modelling language

The simple modelling language is a fragment of UML's or MOF's class diagrams, with only three notions: *class*, *attribute*, and *function*. The benefit of treating such a limited language is that the semantics can be worked out in full detail without becoming too lengthy.

A *class* is the extension of a concept of the application domain;[4] and the first category of the simple modelling language is 'class'.

[4] This, and other explanations of UML concepts, serve only to guide the modelling process. They have no impact on the formal treatment. The use of the word *class* in logic originates with Peano who defines it as an "aggregation of entities" [28, p. x].

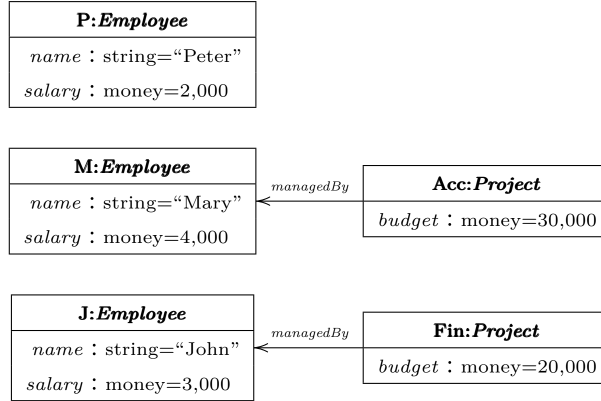


Figure 5 The instance ep of the model EP in the simple modelling language. The names of the instances are written before the class names, the values of the attributes are written after their declarations, and the value of a function at an instance is indicated by an arrow.

An *attribute* of a class is a characteristic applicable to every object in the extension of the class. Each attribute of a class is typed by a datatype drawn from the set D, called the *value type* of the attribute. For any given object of the class, the value of the attribute is of this type. The second category of the simple modelling language is $\mathrm{attrib}(A, \Gamma)$, where $A : \mathrm{class}$ and $\Gamma : \mathrm{D}$.

A *function* from one class to another is an assignment of exactly one object of the second class to each object of the first class. The third category of the simple modelling language is $f : \mathrm{fun}(A, B)$. The classes A and B will be called, respectively, the *domain* and *value* classes of the function f.

A *model* is a sequence of uninterpreted symbols (variables) declared to be of categories of the language, i.e., a telescope [12]. The categories of a model have to be well-formed in virtue of previously introduced uninterpreted symbols.[5] Thus, in general, a model has the form

$X_1 : \mathrm{class}, \ldots, X_m : \mathrm{class},$
$Y_1 : \mathrm{attrib}(X_{c_1}, \gamma_1), \ldots, Y_n : \mathrm{attrib}(X_{c_n}, \gamma_n),$
$Z_1 : \mathrm{fun}(X_{d_1}, X_{v_1}), \ldots, Z_p : \mathrm{fun}(X_{d_p}, X_{v_p}),$

where the symbols X_i are distinct, as are Y_i and Z_i; moreover, $1 \le c_i, d_i, v_i \le m$, and $\gamma_i : \mathrm{D}$. If needed, this can be encoded in type theory by

$$M = \sum_{(X,Y,Z) : \mathrm{enum}^3} \{c : X^Y, \gamma : \mathrm{D}^Y, d : X^Z, v : X^Z\}, \tag{6}$$

where 'enum' is the set of finite collections of names, the curly braces denote a standard record type, and X^Y means the same as $Y \to X$.

A *class diagram* (Figure 3) is the representation of a model as boxes and arrows according to the correspondence explained above. From this point onwards, the class diagram and the formal notation for the model will be considered interchangeable — as two expressions of the same thought.

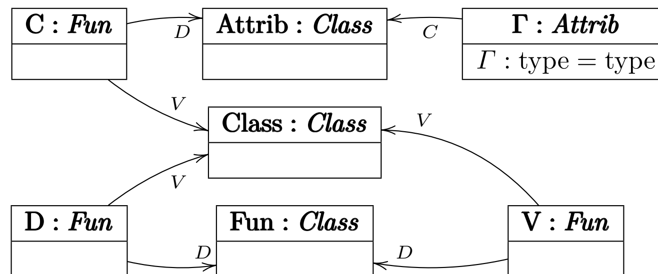


Figure 6 The instance u of the universal model U with the property that $\rho(\mathrm{u}) = \mathrm{U}$ and $\pi(\mathrm{U}) = \mathrm{u}$. Compare with Figure 4.

This formalisation of the notion of class diagram means, in

[5] So that, e.g., a class has to be introduced before its attributes, and the domain and value classes of a function have to be introduced before the function. Cf., the notion of *context* [16, 32].

particular, that it is easy to decide whether a given diagram is well-formed or not: simply write down the corresponding model and make sure it is well-formed.

An *instance* i of a model m is an interpretation of its uninterpreted symbols according to the following scheme:[6]

(1) a class symbol A : class is interpreted by a finite set A^i;
(2) an attribute symbol a : attrib(A, Γ) is interpreted by a function $a^i : A^i \to \mathrm{T}(\Gamma)$;
(3) and a function symbol f : fun(A, B) is interpreted by a function $f^i : A^i \to B^i$.

Note that there is at least one instance of any model, viz., the empty instance, in which all class symbols are interpreted by the empty set, and all attribute and function symbols by the "empty" function (from the empty set).

Instances can also be displayed as diagrams. For example, the instance ep (Figure 5) of the model EP (Figure 3) is defined as follows:

$\mathit{Employee}^{\mathrm{ep}} = \{\mathrm{P}, \mathrm{M}, \mathrm{J}\}$,
$\mathit{Project}^{\mathrm{ep}} = \{\mathrm{Acc}, \mathrm{Fin}\}$,
$\mathit{name}^{\mathrm{ep}} = \{\mathrm{P} \mapsto \text{“Peter”}, \mathrm{M} \mapsto \text{“Mary”}, \mathrm{J} \mapsto \text{“John”}\}$,
$\mathit{salary}^{\mathrm{ep}} = \{\mathrm{P} \mapsto 2,000, \mathrm{M} \mapsto 4,000, \mathrm{J} \mapsto 3,000\}$,
$\mathit{budget}^{\mathrm{ep}} = \{\mathrm{Acc} \mapsto 30,000, \mathrm{Fin} \mapsto 20,000\}$,
$\mathit{managedBy}^{\mathrm{ep}} = \{\mathrm{Acc} \mapsto \mathrm{M}, \mathrm{Fin} \mapsto \mathrm{J}\}$.

Encoded in type theory, the set of instances of a given model is defined by

$$I((X, Y, Z), \{c, \gamma, d, v\}) = \sum_{|\cdot| :\, X \to \mathrm{enum}} \left(\prod_{y :\, Y} |c(y)| \to \gamma(y) \right) \times \left(\prod_{z :\, Z} |d(z)| \to |v(z)| \right). \quad (7)$$

Recall that Σ and Π stand for disjoint union and Cartesian product of indexed families of sets.

6 A universal model for the simple modelling language

A universal model, written U, of the simple modelling language is presented in Figure 4. It corresponds to the following sequence of assertions:

Class : class,
Attrib : class,
Fun : class,
Γ : attrib$(\mathit{Attrib}, \mathrm{type})$,
C : fun$(\mathit{Attrib}, \mathit{Class})$,
D : fun$(\mathit{Fun}, \mathit{Class})$,
V : fun$(\mathit{Fun}, \mathit{Class})$.

▶ Theorem. The simple modelling language described in Sect. 5 is self-describing.

Proof. We must show that U is a universal model, i.e., we must define ρ and π and show that they are inverse of each other. Let s be a instance of the model U. Assume that

$\mathit{Class}^s = \{A_1, \ldots, A_m\}$,
$\mathit{Attrib}^s = \{a_1, \ldots, a_n\}$,
$\mathit{Fun}^s = \{f_1, \ldots, f_p\}$,
$\Gamma^s : \mathit{Attrib}^s \to \mathrm{T}(\mathrm{type})$,
$C^s : \mathit{Attrib}^s \to \mathit{Class}^s$,

[6] Using the terminology of logic, a model is an uninterpreted language and an instance is an interpretation of its uninterpreted symbols. Cf. [31] and [2].

$D^s : \mathit{Fun}^s \to \mathit{Class}^s$,
$V^s : \mathit{Fun}^s \to \mathit{Class}^s$.

Recall that a model is a sequence of uninterpreted symbols declared to be of certain categories. The model $\rho(s)$ is defined as follows:

$A_1 : \text{class}, \ldots, A_m : \text{class}$,
$a_1 : \text{attrib}(C^s(a_1), \mathit{\Gamma}^s(a_1)), \ldots, a_n : \text{attrib}(C^s(a_n), \mathit{\Gamma}^s(a_n))$,
$f_1 : \text{fun}(D^s(f_1), V^s(f_1)), \ldots, f_p : \text{fun}(D^s(f_p), V^s(f_p))$.

This model is always well-formed in the sense described above, i.e., symbols are unique within each form of category (class, attrib, and fun).

Conversely, let S be a model of the simple modelling language, given by

$B_1 : \text{class}, \ldots, B_m : \text{class}$,
$b_1 : \text{attrib}(B_{c_1}, \gamma_1), \ldots, b_n : \text{attrib}(B_{c_n}, \gamma_n)$,
$g_1 : \text{fun}(B_{d_1}, B_{v_1}), \ldots, g_p : \text{fun}(B_{d_p}, B_{v_p})$,

where $\gamma_1, \ldots, \gamma_n$ are elements of the set $\mathrm{D} = \mathrm{T}(\text{type})$, and each of the numbers $c_1, \ldots, c_n$, $d_1, \ldots, d_p$, and $v_1, \ldots, v_p$ are in the range $1, \ldots, m$. Then $\pi(S)$ is an instance of U given by

$\mathit{Class}^{\pi(S)} = \{B_1, \ldots, B_m\}$,
$\mathit{Attrib}^{\pi(S)} = \{b_1, \ldots, b_n\}$,
$\mathit{Fun}^{\pi(S)} = \{g_1, \ldots, g_p\}$,
$\mathit{\Gamma}^{\pi(S)}(b_x) = \gamma_x \ : \mathrm{T}(\text{type})$,
$C^{\pi(S)}(b_x) = B_{c_x} \ : \mathit{Class}^{\pi(S)}$,
$D^{\pi(S)}(g_y) = B_{d_y} \ : \mathit{Class}^{\pi(S)}$,
$V^{\pi(S)}(g_y) = B_{v_y} \ : \mathit{Class}^{\pi(S)}$.

To show that $\pi(\rho(s)) = s$, let s and S be defined as above, and consider $\pi(\rho(s))$, where $S = \rho(s)$. Comparing the definition of S with the definition of $\rho(s)$, we get $A_i = B_i$ (as symbols), $a_i = b_i$, $f_i = g_i$, $C^s(a_x) = B_{c_x}$, $\mathit{\Gamma}^s(a_x) = \gamma_x$, $D^s(f_y) = B_{d_y}$, and $V^s(f_y) = B_{v_y}$. The result follows from a comparison with the definition of $\pi(S)$.

To show that π is also a right inverse of ρ, let S be given as above and plug $\pi(S)$ into the definition of ρ. The result is S. ◀

An obvious use of this Theorem is to apply the function π to the model U. The resulting instance, Figure 6, should be studied carefully. It is also instructive to compare it with Table 2.

Figure 7 shows the reification of the diagram of Figure 3.

7 A less simple modelling language

This Section is deliberately brief, and many details are left to the reader. It is best viewed as an extended example of how to apply the techniques introduced earlier in the paper. The example is based on Figure 9, showing the universal model of a significant fragment of the class diagrams of xUML [25].[7] The main differences between this less simple language and the previously introduced simple language are outlined below.

First, there is one more datatype, viz., 'mult', of multiplicities, i.e.,

$$\begin{aligned} \mathrm{D} &= \{\text{string}, \text{money}, \text{type}, \text{mult}\}, \\ \mathrm{T}(\text{mult}) &= \{a..b \mid a : \mathrm{N}, b : \mathrm{N} \cup \{\star\}, a \leq b\}, \end{aligned} \tag{8}$$

[7] xUML is a fragment of UML that is designed to facilitate the *execution* of models.

Model	Instance i
A : class	A^i : set
a : attrib(A, Γ)	$a^i : A^i \to \mathrm{T}(\Gamma)$
R : assoc(A, B)	$R^i : A^i \times B^i \to$ prop
r : rrole(A, B, R, o)	$r_1^i(x) : o$, $r_2^i(x) : r_1^i(x) \hookrightarrow B^i$, $r_3^i(x)(y) : R^i(x, y) \leftrightarrow (\exists z : r_1^i(x)) r_2^i(x)(z) = y$
l : lrole(A, B, R, λ)	$l_1^i(y) : \lambda$, $l_2^i(y) : l_1(y) \hookrightarrow A^i$, $l_3^i(y)(x) : R^i(x, y) \leftrightarrow (\exists z : l_1^i(y)) l_2^i(y)(z) = x$
e : ident(A, a, Γ)	$e^i : \mathrm{T}(\Gamma) \to A^i + \{\star\}$, s.t. $e^i(x) = \mathrm{left}(y)$ iff $a^i(y) = x$
g : gen$(A, S_1, \ldots, S_n)$	$g^i : A^i \cong S_1^i \times \cdots \times S_n^i$
s : assclass(C, A, B, R)	$s^i : C^i \cong (\Sigma\ (x, y) : A^i \times B^i) R^i(x, y)$

Table 3 The forms of assertion of the less simple modelling language, together with their interpretations in an instance.

where $a..b$ is the set $\{a, a+1, \ldots, b\}$ if b is finite, and $a..\star$ stands for $\{a, a+1, \ldots\}$. The datatypes 'string' and 'money' are as before, and 'type' is still universal.

Table 3 lists the forms of assertions used when translating a less simple diagram to linear notation, together with their interpretations in an instance. Classes and attributes work exactly as for the simple modelling language.

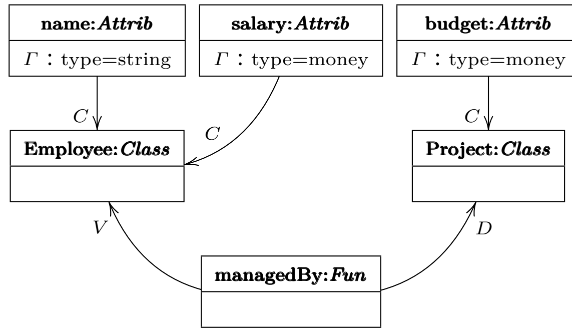


Figure 7 The instance ep of the universal model U with the property that $\rho(\mathrm{ep}) = \mathrm{EP}$ and $\pi(\mathrm{EP}) = \mathrm{ep}$. Compare with Figure 3.

Instead of functions, the less simple modelling language uses associations, which may have two kinds of roles: left and right. An association R : assoc(A, B) is interpreted in type theory by a binary relation R^i on A^i and B^i. A right role r : rrole(A, B, R, o), where o is a multiplicity, is interpreted as a triple valued function $r^i(x) = (r_1^i(x), r_2^i(x), r_3^i(x))$, where $x : A^i$. The first component $r_1^i(x) : o$ gives the multiplicity of x; the second component $r_2^i(x) : r_1^i(x) \hookrightarrow B^i$ is an injection of the multiplicity into B^i;[8] the third component is a proof that an element y of B^i is related to x by R^i if and only if y is in the image of $r_2^i(x)$. Another way to put it is that $r^i(x)$ identifies the subset of B^i, with a finite cardinality drawn from the set o, that is related by R^i to $x : A^i$. Left roles are treated analogously to right roles.

As a special case, when the multiplicity is $o = 1..1$, a right role induces a normal function $A^i \to B^i$. The virtue of this treatment of roles is that it is compositional, i.e., a left or right role can be added to a diagram without changing the interpretation of the original diagram. In fact, formally, nothing prevents an association from having several left or right roles.

Identifiers in xUML serve the same purpose as unique keys in relational databases, i.e., they make it possible to retrieve an instance (row or tuple in database parlance) from the value of an attribute. For example, if there were an identifier of the *name* attribute of the *Employee* class of Figure 3, names would have to be unique, and it would be possible to retrieve the instance corresponding to a name, if any.

[8] Here the number $r_1^i(x)$ is identified with the set on $r_1^i(x)$ elements.

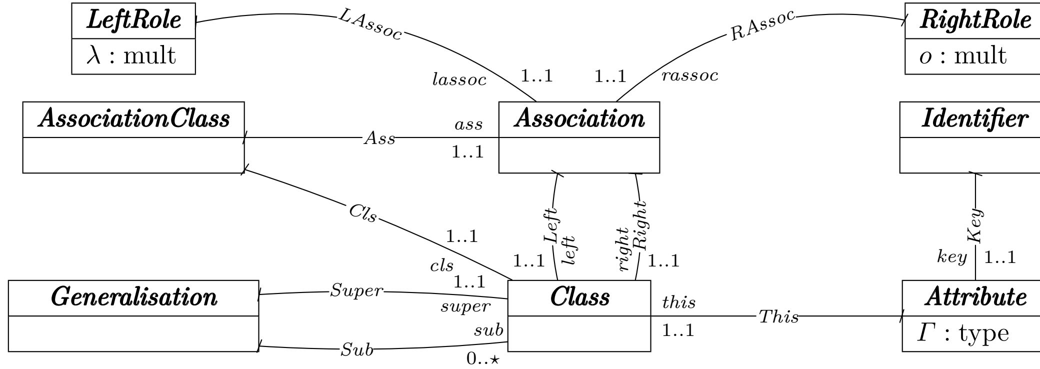


Figure 9 The universal model of a fragment of the modelling language xUML, capable of expressing the notions class, attribute, association, generalisation, association class, identifier, and left and right role.

An identifier $e : \mathrm{ident}(A, a, \Gamma)$ of an attribute a indicates that the values of the attribute are different for different instances of the class A.[9] The identifier e is interpreted in an instance i as a function e^i from $\mathrm{T}(\Gamma)$ to the set $A^i + \{\star\}$, such that e^i is a partial inverse of a^i, i.e., for all $x : A^i$ and $y : \mathrm{T}(\Gamma)$, $e^i(x) = \mathrm{inl}(y)$ if and only if $a^i(y) = x$. Here 'inl' denotes the canonical injection $A^i \hookrightarrow A^i + \{\star\}$.

A generalisation $g : \mathrm{gen}(A, S_1, \ldots, S_n)$ is interpreted in an instance i as an isomorphism between the interpretation of the superclass A^i and the interpretations of its subclasses $S_1^i \times \cdots \times S_n^i$.

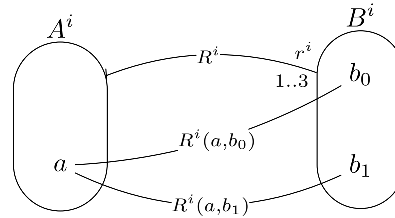


Figure 8 The interpretation a right role $r : \mathrm{rrole}(A, B, R, 1..3)$ in an instance i, where A^i has an element a related to exactly two elements b_0 and b_1 of B^i. In particular, $r_1^i(a) = 2$, and $r_2^i(a) : \{0, 1\} \hookrightarrow B$, with $r_2^i(a)(j) = b_j$.

An association class $s : \mathrm{assclass}(C, A, B, R)$ between a class C and an association R is interpreted as an isomorphism between C^i and the set of pairs (x, y) in $A^i \times B^i$ that are related by R^i.

8 Related work

There are several approaches to the semantics of UML and MOF class diagrams, e.g., logic based [3], graph based [33], coinductive [29], or, like this paper, algebraic [5, 13].

Our modelling languages depart from the MOF in two important respects. We consider generalisation instead of inheritance; and, as opposed to UML and MOF, we have no common genus of datatypes and classes.

Generalisation and inheritance are sometimes taken as synonymous, but I think there is

[9] This paper makes a significant departure from xUML identifiers (and database uniqueness constraints) by only allowing one attribute to participate in an identifier; a faithful encoding would require the multiplicity of the role *key* of Figure 9 to be one to many. However, if the multiplicity was simply changed, the model of Figure 9 would no longer be universal, as instances would include identifiers combining several attributes of different classes. Thus, the modelling language would have to be significantly strengthened to cater for identifiers with higher multiplicity.

an important distinction to be made. By inheritance, I mean the relation B *inherits from* A, that would be interpreted by $B^i \subset A^i$ in an extensional framework. This is difficult to formalize in type theory as there is no subset relation. However, the relation B *is generalised by* A can be interpreted by an injection $B^i \hookrightarrow A^i$.

As regards the existence of a common genus of datatypes and classes, it is interesting to review what Date [9, p. 865] calls *the great blunder*. There are three notions involved: the notion of *datatype*, i.e., our D or what Date calls *domain*; the notion of relational variable (*relvar* in relational database theory); and the notion of *class*. Date's main point is that *datatype* $\neq$ *relvar*, and this distinction is maintained in this paper. In fact, our notion of class is similar to the notion of *relvar* — to begin with, both are variables.

However, what Date actually calls *the great blunder* is the identification *relvar* = *class* (made here): that is, he considers the identification *datatype* = *class* correct. Date's identification is based on the conception of a class as a record type.

In this paper, the notion of class is identified with the notion of relvar (rather than with the notion of datatype) because object-oriented programming is based on the idea that a program can create a *new* instance of a class. The classes of this paper support the *new* operation, and relvars support the *insert* operation: in both cases, one element is added to the set interpreting the variable. Datatypes, on the other hand, are more like mathematical sets, and, e.g., the idea of creating a *new* number is repugnant. To conclude, this paper makes *the great blunder* in words, but not in spirit.

My approach to the translation of model diagrams into type theory differs from that of Poernomo et al. [29, 14] in one important respect: type-theoretic concerns have influenced my design of the modelling languages, while Poernomo et al. have taken the MOF at face value. Encoding the full MOF requires coinductive datatypes and definitions by corecursion, which soon lead to rather complex formalisations. In addition, the semantics becomes noncompositional, due to the outermost fixpoint operator in the definition of models. I have avoided these problems by simplifying the modelling language.

An analog to the notion of class diagram, with respect to how its semantics has evolved from a mere "blackboard" semantics, is the notion of state chart, as expounded by Harel [20]. A precise constructive semantics for a species of state charts is given by André [1].

9 Conclusion and future work

In my opinion, one of the main obstacles to model driven approaches gaining wide acceptance in the industry is insufficient tool support. One step in the right direction would be to formalize the simple (or less simple) modelling language inside a proof assistant like Coq [4] or Agda [27].

In addition to allowing formal manipulation of models, such a tool could make it possible to generate a diagram from a possibly annotated model instance, thus reinforcing the point that diagrams are valid formal expressions and, with time, changing a view held by many software engineers, viz., that diagrams are inherently vague [31].

The reader may have noticed that the two modelling languages presented in this paper, although using the notation of UML class diagrams, are semantically more akin to the entity-relationship model [6] or ORM [18]. It would be interesting to find out what characterises features of data modelling and object-oriented programming that can be interpreted using the direct approach of Sect. 4.

A difference between Figure 4 and Figure 9 is that, in the forme, all features of the modelling language are used to define the universal model, whereas, in the latter, the four notions

class, *attribute*, *association*, *right role* would suffice. That is, the notions *generalisation*, *association class*, *left role*, and *identifier* are like appendices to a smaller modelling language. Does a modelling language with an irreducible universal model have any advantage over a modelling language with redundant features?

One potential direct application of the simple modelling language is as a data model for a non-relational database management system, using the identification *database schema* = *model*. Several database maintenance operations could be simplified by using the universal model U. For example, to define a new database schema one would simply have to define an instance of U. This definition would use the same syntax as the definition of an instance of any other model.

In this context, it would also be interesting to consider how data manipulation operations interact with ρ and π. For example, creating a new instance of the class *Class* in an instance of U could create a new class in the corresponding model.

Acknowledgements

This work was partially supported by Engineering and Physical Sciences Research Council (EPSRC) grant number EP/G03012X/1.

Thanks to I. Poernomo for teaching me about metamodelling and the MOF. Thanks also to P. Martin-Löf, E. Palmgren, O. Wilander, and other participants of the Stockholm-Uppsala logic seminar for valuable comments on an early version of this paper.

Moreover, I thank D. Calvanese, I. Feinerer, T. Halpin, D. Harel, G. Karsai, S. Mellor, B. Rumpe, and the anonymous reviewers for valuable corrections, amendments, and comments to a draft version. Finally, thanks to the anonymous reviewers for valuable corrections and comments.

References

1 C. André. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science*, 88:3–19, 2004.
2 C. Atkinson and T. Kühne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.
3 D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1–2):70–118, 2005.
4 Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
5 A. Boronat and J. Meseguer. An algebraic semantics for MOF. In J.L. Fiadeiro and P. Inverardi, editors, *FASE 2008*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008.
6 P. P.-S. Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
7 E. F. Codd. *The Relational Model for Database Management*. Addison-Wesley, 1990.
8 T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.
9 C. J. Date. *An Introduction to Database Systems*. O'Reilly, 7 edition, 2000.
10 C. J. Date. *Database in Depth*. O'Reilly, 2005.
11 M. Davis. *Engines of Logic: Mathematicians and the Origin of the Computer*. W. W. Norton & Company, NY, 2000.
12 N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Inform. Comput.*, 91(2):189–204, 1991.

13 I. Feinerer and G. Salzer. Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints. In *TASE 2007*, pages 411–420. IEEE Computer Society Press, 2007.
14 C. Fiorentini, A. Momigliano, M. Ornaghi, and I. Poernomo. A constructive approach to testing model transformations. In L. Tratt and M. Gogolla, editors, *ICMT 2010*, volume 6142 of *LNCS*, pages 77–92. Springer, 2010.
15 J. Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.
16 J. G. Granström. *Treatise on Intuitionistic Type Theory.* Logic, Epistemology, and the Unity of Science. Springer, 2011.
17 J. G. Granström. A new paradigm for component-based development. *Journal of Software*, 7(5):1136–1148, 2012.
18 T. A. Halpin. Object-role modeling: principles and benefits. *IJISMD*, 1(1):33–57, 2010.
19 P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *LNCS*, pages 317–331, 2000.
20 D. Harel. Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
21 D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.
22 P. Hoogendijk and O. de Moor. Container types categorically. *J. Funct. Program.*, 10(2):191–225, 2000.
23 ISO/IEC 9075-11:2008. Information and definition schemas (SQL/schemata). Technical report, ISO, Geneva, Switzerland, 2008.
24 P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory. Bibliopolis, Napoli, 1984.
25 S. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture.* Addison Wesley, 2002.
26 A. Møller. Document structure description. Technical report, BRICS, 2005.
27 U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2009.
28 G. Peano. *Arithmetices Principia Nova Methodo Exposita.* Fratelli Bocca, Turin, 1889.
29 I. Poernomo. The meta-object facility typed. In *SAC*, pages 1845–1849, 2006.
30 D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.
31 E. Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003.
32 A. Tasistro. *Formulation of Martin-Löf's type theory with explicit substitutions.* Licentiate thesis, Chalmers University of Technology, 1993.
33 X. Thirioux, B. Combemale, X. Crégut, and P. L. Garoche. A framework to formalise the MDE foundations. In R. Paige and J. Bézivin, editors, *International Workshop on Towers of Models*, pages 14–30, 2007.
34 A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.
35 N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

Testing versus proving in climate impact research

Cezar Ionescu[1] and Patrik Jansson[2]

1 **Potsdam Institute for Climate Impact Research**
Telegrafenberg A31, 14473 Potsdam, Germany
`ionescu@pik-potsdam.de`

2 **CSE Department, Chalmers University of Technology**
SE - 412 96 Göteborg, Sweden
`patrikj@chalmers.se`

Abstract

Higher-order properties arise naturally in some areas of climate impact research. For example, "vulnerability measures", crucial in assessing the vulnerability to climate change of various regions and entities, must fulfill certain conditions which are best expressed by quantification over all increasing functions of an appropriate type. This kind of property is notoriously difficult to test. However, for the measures used in practice, it is quite easy to encode the property as a dependent type and prove it correct. Moreover, in scientific programming, one is often interested in correctness "up to implication": the program would work as expected, say, if one would use real numbers instead of floating-point values. Such counterfactuals are impossible to test, but again, they can be easily encoded as types and proven. We show examples of such situations (encoded in Agda), encountered in actual vulnerability assessments.

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, D.1.6 Logic Programming, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging

Keywords and phrases dependently-typed programming, domain-specific languages, climate impact research, formalization

Digital Object Identifier 10.4230/LIPIcs.TYPES.2011.41

1 Introduction

Climate impact research is not the same as climate research: it does not deal, for example, with building the detailed simulations of the climate system that run on massively parallel machines of incredible, yet always insufficient computational power. Rather, climate *impact* research attempts to analyze the broad, first-order effects of various policies meant to mitigate or alleviate the problems caused by human-induced climate change. The Potsdam Institute for Climate Impact Research (the acronym *PIK* comes from the more compact German version: *Klimafolgenforschung*) has on its web page the following introduction:

> At PIK researchers in the natural and social sciences work together to study global change and its impacts on ecological, economic and social systems. They examine the Earth system's capacity for withstanding human interventions and devise strategies for a sustainable development of humankind and nature.
> PIK research projects are interdisciplinary and undertaken by scientists from the following Research Domains: Earth System Analysis, Climate Impacts and Vulnerabilities, Sustainable Solutions and Transdisciplinary Concepts and Methods.
> Through data analysis, computer simulations and models, PIK provides decision makers with sound information and tools for sustainable development. In addition to publishing results in scientific journals the Institute gives advice to national and regional authorities and, increasingly, to global organisations such as the World Bank.
>
> (from `http://www.pik-potsdam.de/institute`)

© C. Ionescu and P. Jansson;
licensed under Creative Commons License BY-ND

18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson, Bengt Nordström; pp. 41–54

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

The important point here is the following: many complex systems are studied together by scientists from many different disciplines. In this kind of enterprise, the concepts that tend to be most used across disciplines have a high intuitive content, which ensures that they are quickly grasped by all the different parties ("vulnerability" will be our running example, but consider also "stability", "resilience", "global change", "sustainable growth", "green path", and so on). The danger is that each party will grasp it in a different way, hence the importance of definitions. In general, the more formal the definition, the less the risk it will be misunderstood (though the chance of being understood might also decrease), and here is where a first connection to logic and computer science appears.

Additionally, such "fulcrum" concepts that leverage our everyday intuitions and help structure the interdisciplinary discourse also provide natural candidates for assessments, for measurement and comparison, which then, in turn, can be used as the basis for "giving advice to national and regional authorities". Many of these assessments are computer-based, and subject to the usual concerns of reuse, genericity, efficiency and correctness (especially important, one would think, when giving advice "to global organizations such as the World Bank").

This is the computer scientists' playground, and the game plan is: formalize the concepts involved in order to be able to write specifications against which to assess program correctness. Do it generically, in order to unify and reuse as much as possible of the existing code. Since the subject is largely mathematical, use a high-level language with an expressive type system, in order to minimize the distance from specification to implementation. Hopefully, the end-result will be a domain-specific language, which will simplify writing the particular sort of programs we started with, while at the same time making their correctness easier to assess.

This paper presents some of the results we obtained while playing this game within the field of (computer-assisted) *vulnerability assessment*. The next section is a whirlwind tour of definitions of vulnerability and the resulting (simplified) Haskell formalization. We then take up the question of correctness: we want to ensure that key conditions are met by an implementation. The first idea, presented in Section 3, is in tune with current software engineering best practices: apply automatic property-based testing (for example, using QuickCheck [8]). It turns out that writing good tests is somewhere between hard and impossible, but proving on paper that the conditions hold is really easy. Therefore, we re-implemented parts of the system in a dependently-typed programming language (Agda[1], [21, 26]) and found that expressing the conditions as types was at least as easy as thinking up good tests, and that convincing the type checker that the conditions were met was at least as easy as implementing those tests. Moreover, things that were impossible before become not even hard. This is presented in Section 4, which raises questions such as: if proving things is so easy, why does it get such a bad reputation? We have an opinion about this, and you can read it in the conclusions.

2 Vulnerability

In the past decade, the concept of "vulnerability" has played an important role in fields such as climate change, food security and natural hazard studies. Vulnerability studies have often been successful in alerting policymakers to precarious situations. The importance of the

[1] The choice of Agda over, say, Coq, was motivated partly by similarity with Haskell (since we could translate our Haskell code-base), partly by aesthetic considerations and by ease of use. Perhaps the largest role was played by the fact that PIK has quite close ties to Chalmers, where Agda was developed.

concept in the particular field of climate change is described, for example, as follows [13]:

> ... Studies based primarily on the output of climate models tend to be characterized by results with a high degree of uncertainty and large ranges, making it difficult to estimate levels of risk. In addition, the complexity of the climate, ecological, social and economic systems that researchers are modeling means that the validity of scenario results will inevitably be subject to ongoing criticism. ... Such criticisms should not be interpreted as questioning the value of scenarios; indeed, there is no other tool for projecting future conditions. What they do, however, is emphasize the need for a strong foundation upon which scenarios can be applied, a foundation that provides a basis for managing risk despite uncertainties associated with future climate changes. This foundation lies in the concept of vulnerability.

No doubt, vulnerability is one of the "fulcrum" concepts mentioned in the introduction and, alerted to the importance of definitions in an interdisciplinary context, we expect this one to be very well defined. Unfortunately, this is only the case if by "well defined" we mean "defined many times". Figure 1 contains a sample of vulnerability "definitions" found in the literature:

- [16]: Vulnerability is defined as the extent to which a natural or social system is susceptible to sustaining damage from climate change. Vulnerability is a function of the sensitivity of a system to changes in climate (the degree to which a system will respond to a given change in climate, including beneficial and harmful effects), adaptive capacity (the degree to which adjustments in practices, processes, or structures can moderate or offset the potential for damage or take advantage of opportunities created by a given change in climate), and the degree of exposure of the system to climatic hazards.
- [28]: The conditions determined by physical, social, economic, and environmental factors or processes, which increase the susceptibility of a community to the impact of hazards.
- [7] Vulnerability, therefore, is a human-induced situation that results from public policy and resource availability/distribution, and it is the root cause of many disaster impacts. Indeed, research demonstrates that marginalized groups invariably suffer most in disasters. Higher levels of vulnerability are correlated with higher levels of poverty, with the politically disenfranchised, and with those excluded from the mainstream of society.
- [6] Vulnerability (in contrast to poverty which is a measure of current status) should involve a predictive quality: it is supposedly a way of conceptualizing what may happen to an identifiable population under conditions of particular risk and hazards. Is the complex set of characteristics that include a person's: initial well-being (health, morale, etc.); self-protection (asset pattern, income, qualifications, etc.); social protection (hazard preparedness by society, building codes, shelters, etc.); social and political networks and institutions (social capital, institutional environment, etc.).
- [9] Vulnerability (V) = Hazard Coping,
  with Hazard = H (Probability of the hazard or process; shock value; predictability; prevalence; intensity/strength);
  and Coping = C (Perception of risk and potential of an activity; possibilities for trade; private trade, open trade).

Figure 1 A sample of vulnerability definitions from several different papers.

There are many, many more such definitions, a large percentage of which wouldn't pass Pascal's requirement of "application of a name to things which are clearly designated by

terms perfectly known" [25]; the curious reader is referred to Thywissen's summary of some thirty-odd definitions [27].

There is a corresponding diversity in the way in which vulnerability is *measured.* Examining the technical details of computer-assisted vulnerability assessments is tedious, but has a clear advantage over reading definitions such as the above: one can unambiguously determine what is being measured.

Virtually all vulnerability assessments have the following structure. First, one tries to estimate the evolution of various parameters of interest, for example, the average temperature in a given region, the gross domestic product of a country, the sea-level of some coastal area, but also less immediately relevant values, such as literacy rate or number of telephone lines in a region [17]. Sometimes, the result of this forecasting analysis is a list of values, one element for each time period (week, month or year) of the time horizon (typically measured in decades). Most times, the result will consist of several such trajectories, perhaps with some additional information about their likelihood. Thus, one can have lists of possible trajectories, or a probability distribution over trajectories, or a fuzzy set of trajectories, etc.

Next, each trajectory is examined in order to determine the harm that befalls the region or population under consideration: damages, negative impacts, losses caused by the factors of interest (for example, human-induced climate change). Harm is represented in many ways, but it is always assumed that the resulting values can be at least partially compared, i.e., that they are members of a preordered set.[2]

Depending on how the forecast of the parameters was achieved, we have so far a list of harm values, or a probability distribution over harm values, or a fuzzy set, etc. Now comes the final step: aggregating all these harm values, obtaining the final vulnerability assessment. This is usually done either by taking some representative value, for example the maximal or the likeliest harm, or by an integral measure of the possibilities (such as their sum or average). The final value does not need to lie in the same set as the harm values, but vulnerability values also need to form at least a preorder: the purpose of the assessment is often to compare the relative vulnerabilities of regions, or of the same region under different scenarios.

In Haskell, these explanations can be expressed more concisely and precisely:

```
data State          = ...                  -- an appropriate type for the values of the
                                           -- parameters of interest
type Trajectory     = [State]              -- a trajectory is a list of states
type Possible       = ...                  -- a functor which represents the structure
                                           -- of possible trajectories, e.g. List
data V              = ...                  -- datatype of harm values
instance Preorder V where ...              -- harm values must be preordered
data W              = ...                  -- datatype for vulnerability values
instance Preorder W where ...              -- vulnerability values must be preordered
vulnerability :: Possible Trajectory ->    -- possible trajectories
                 (Trajectory -> V)   ->    -- harm evaluation
                 (Possible V -> W)   ->    -- aggregation of harm values
                 W                         -- type of final result
vulnerability possible harm measure = measure (fmap harm possible)
```

[2] The reason for not requiring anti-symmetry is that harm values are often compared via cost functions.

Possible trajectories are collected together in a functorial structure. Besides the fact that all our examples (probability distributions, fuzzy sets, lists) are functors, this makes sense because of the need to apply the harm evaluation function to each trajectory. Otherwise, the code follows literally the description above.

Most of the work in a vulnerability assessment is put in computing the structure of possible trajectories. To do this, existing models are used (and reused), which are usually written by specialists in the relevant disciplines: economists, climate scientists, geographers, social scientists, etc. The models are then combined by the team that does the vulnerability assessment. Sometimes, these models have different types: a climate model might yield a deterministic trajectory of the average global temperature, while a demographic model might offer only a list of possible evolutions of the population, and an economic model a probability distribution over possible future values of the gross domestic product. Accordingly, most of the work we have done was in extracting the general structure of these models and of the means of combining them, in order to simplify the task of the vulnerability assessment in its most difficult part. The result was a domain-specific language for describing and combining *monadic dynamical systems*, described extensively by Ionescu [11] and concisely by Lincke et al. [14].

Here, however, we concentrate on the computationally less intensive part: the interplay between the evaluation of harm and the measurement of vulnerability. There is very little one can say to better describe the possible candidates for these functions: one cannot claim, for example, that only certain preordered sets are suitable and exclude others. But there is a condition which virtually everybody agrees on: if the harm evaluations along all trajectories in a structure are increased, then the vulnerability measure should also increase. This kind of monotonicity can be taken as the defining condition for a vulnerability measure:

▶ **Definition 1.** *Let V and W be two preorders, and F a functor. A function $m : F\ V \to W$ is called a* vulnerability measure *if, for any increasing function $i : V \to V$ (that is, $v \leqslant i\ v$ for all $v : V$), and any $x : F\ V$ we have $m\ x \leqslant m\ (F\ i\ x)$.*

If we use the order $x \sqsubseteq_m y = m\ x \sqsubseteq m\ y$ on $F\ V$ we can say that m is a vulnerability measure if "$(F\ i)$ is increasing when i is increasing". We will use this formulation in Section 4. No matter how good the models used to forecast the possible trajectories are, no matter how well combined, if a vulnerability assessment uses a function which is not a vulnerability measure in order to aggregate the harm values, then it must be regarded as flawed.

Are there any vulnerability assessments which fail in this respect? Unfortunately, yes. The "likeliest harm value" we mentioned above does not fulfill this condition, and neither do other "democratic" methods (the most frequent result of harm values, for instance). There is, therefore, scope for error, and so we come to the idea of *testing*, for a given implementation, that the vulnerability measure condition holds.

3 Testing vulnerability measures

To test a candidate vulnerability measure $m : F\ V \to W$ we first turn to the question of the functoriality of the structure of type $F\ V$ that collects the harm values. How do we know that the implementation of the mapping function preserves identities and compositions? The Haskell type system does not detect the problem with

```
mapTry :: (a -> b) -> [a] -> [b]
mapTry f []       = []
mapTry f (a : as) = mapTry f as
```

The problem is that $mapTry\ id\ = const\ [\,] \neq id$, so the first functor law fails (but the second functor law holds). As an aside, *mapTry* is the version suggested by Agda's automatic theorem prover / type inhabitant searcher, called Agsy [15]. (To Agsy's defence should be said that it only aims at, and succeeds in, finding *some* value of the correct type.)

If we want to test if polymorphic properties like the functor laws hold for a polymorphic function like *mapTry*, we need to pick some monomorphic type to test them on. It is not in general enough to pick a trivial type like () or a small type like *Bool*, but most often it is enough to test with the type of natural numbers. For the functor laws the results of Bernardy et al. [2] allow us to reduce testing the polymorphic map function to just one type (and in fact, just we can even fix the function argument f), but there is still the question of coverage:

```
map :: (a → b) → [a] → [b]
map f []       = []
map f (a : as) = if length as ⩾ bigNumber
                 then map f as
                 else f a : map f as
```

Granted, this is a malicious example, but the problem remains, especially in the case of functors that require more complex implementations (such as the simple probability functor). Still, let us accept for now that the implementation of the mapping function is likely to be used in many programs and therefore verified in so many different cases that we can take it to be correct.

For concreteness, let us fix the functor to be the non-empty list functor given by

```
data List a = Wrap a | Cons a (List a)
  deriving (Ord, Eq, Show)

fold :: (a → b) → (a → b → b) → List a → b
fold w c (Wrap a)    = w a
fold w c (Cons a as) = c a (fold w c as)

instance Functor List where
  fmap f = fold (Wrap ∘ f) (λ a bs → Cons (f a) bs)
```

A typical type for harm values is a tuple: pairs of floating-point numbers representing (monetary) damages and natural numbers representing lost lives. The least controversial way of comparing such values is given by the dominance relation:

```
instance POrd a where
  leq :: a → a → Bool

instance (POrd a, POrd b) ⇒ POrd (a, b) where
  (a1, b1) `leq` (a2, b2) = a1 `leq` a2 ∧ b1 `leq` b2
```

We defined a new type class for preorders, similar to the *Ord* class provided by Haskell. Instances of the Haskell *Ord* class are required to be total orders, while instances of *POrd* should be preorders. Neither of these requirements can be expressed in Haskell, so there is no automatic check that instances really satisfy them. Anyway, let us grant that the preorder properties also do not need to be tested here (either because they are tested elsewhere, or because the implementation can be trivially seen to be correct).

The biggest problem that we encounter in testing vulnerability measures is its higher-order nature, namely the quantification over all possible increasing functions. In QuickCheck notation, one might write

```
testMonotonicity m i x = increasing i ==> m x `leq` m (fmap i x)
```

This naive translation of the requirement would check that i is an increasing function, and then check that v assigns an increased measure to the increased x. Even assuming the unlikely case in which the property of being increasing is decidable (this only works for functions with finite domain – not the case in our example), we still have the problem that arbitrarily generated functions are unlikely to be increasing, and QuickCheck will stop with an inconclusive result once it reaches the maximum number of attempts for which it is configured.

Thus, we need to use a custom generator which guarantees that the functions it generates are increasing:

```
testMonotonicity m genInc x = forAll genInc (λ i →
                                  m x `leq` m (fmap i x))
```

The problem of coverage will still stay with us, but at least we can ensure that we reach the test of m. For the concrete example we have taken, we can, for example, implement a custom generator by:

```
genInc :: Gen ((Float, Int) → (Float, Int))
genInc = do dx ← choose (0, 10)
            dn ← choose (0, 10)
            return (λ (x, n) → (x + dx, n + dn))
```

and, in fact, we have done so [11]. Unfortunately, this can cause an error: large integers can overflow and result in large negative integers. To do a proper job, the generator has to examine its arguments, and make sure that the returned values really fulfill the desired condition.

Even with the best generator, we still have a problem. Consider a measure which just sums up the elements of the list of potential results:

```
sumList ::  List (Float, Int) → (Float, Int)
sumList = fold id f
  where f (x, n) (x′, n′) = (x + x′, n + n′)
```

This should be a vulnerability measure: increasing the values in a list increases their sum. However, testing it can again fail if the integral part overflows, or if summing up the floating point leads to round-off errors. This means that we need to control also the generation of the arguments, not just the generation of the increasing functions. This is particularly annoying, considering that an alternative popular measure, taking the maximal elements on components, has the same structure as summing the values:

```
supList ::  List (Float, Int) → (Float, Int)
supList = fold id f
  where f (x, n) (x′, n′) = (max x x′, max n n′)
```

The similarity of their names reflects the similarity of their implementations: both functions are folds, the only difference being the use of *max* instead of +. Nevertheless, we cannot with impunity use the generators for *supList* when testing *sumList*. Moreover, in writing more and more complicated generators, we mix up the test for the "interesting" monotonicity condition, with the "implementational" defending against overflow or round-off errors. And we still have a coverage problem, because only with knowledge of the implementation of

the measure can we estimate how well the sampling of the space of increasing functions is achieved.

It might be thought that we can always get around implementational aspects by choosing better representations for numerical values. For example, we can avoid round-off errors by replacing *Float* with rational numbers. Unfortunately, we cannot do that if the vulnerability measure requires computations which cannot be carried out on rational numbers, such as the geometric mean. Resorting to exact real numbers does not solve our problem either, because the order relation on these is not decidable, and we just trade one type of interference from the implementational aspects (defending against round-off errors) for another (guarding against undecidable comparisons).

To sum up:

- We need detailed analysis of the implementation of the function under test, and, in particular, of the datatypes they act on.
- We often need to write different custom generators even for very similar cases (such as *sumList* and *supList*).
- We mix the conceptual part of the tests with the implementational part.
- Good coverage is hard to achieve.

4 Proving correctness of vulnerability measures

It is tempting to point an accusing finger at the higher-order nature of the formalization of the vulnerability measure condition. If we hadn't used Haskell, with its functional nature and expressive type system, we might not have run into so much trouble testing the resulting implementations. Testing higher-order functions is not a topic in common textbooks on software testing [1, 20].

On the other hand, thinking about the problems we saw in the discussion of testing functoriality, it might just be that the culprit is not the exaggerated expressivity of Haskell, but on the contrary: the fact that it is not expressive enough!

In a dependently-typed programming language such as Agda, we can formulate the functor laws as types via the Curry-Howard isomorphism[3]:

```
_≐_ : {A B : Set} → (f g : A → B) → Set
f ≐ g = ∀ a → f a ≡ g a

record Functor (F : Set → Set) : Set1 where
  field
    fmap    : {A B : Set} → (A → B) → F A → F B
    idLaw   : {A : Set} →
              fmap (id {A}) ≐ id {F A}
    compLaw : {A B C : Set} → (f : B → C) → (g : A → B) →
              fmap (f ∘ g) ≐ (fmap f ∘ fmap g)
```

Now we can also prove that the mapping function we defined is indeed functorial. The implementation of non-empty lists is virtually identical to the Haskell version:

[3] We use everywhere the propositional equality type (_≡_) provided by Agda as if it were the only equivalence relation of interest. Parameterising by different equivalence relations (using setoids instead of sets) does not introduce difficulties, but makes the examples more tedious and wastes space. Similar remarks apply to universe-polymorphism.

```
data List (A : Set) : Set where
  [_]  : A → List A
  _::_ : A → List A → List A

fold : {A B : Set} → (A → B) → (A → B → B) → List A → B
fold w c [a]        = w a
fold w c (a :: as)  = c a (fold w c as)

map : {A B : Set} → (A → B) → (List A → List B)
map f = fold ([_] ∘ f) (λ a bs → f a :: bs)
```

Proving that the map function defined preserves identities and composition is actually almost entirely performed by Agsy, the only nudging it needed was to "use the congruence of something" in the inductive step.

```
mapId : {A : Set} →
          map (id {A}) ≐ id
mapId [a]        = refl
mapId (a :: as)  = cong (λ as → a :: as) (mapId as)

mapComp : {A B C : Set} → (f : B → C) → (g : A → B) →
            map (f ∘ g) ≐ (map f ∘ map g)
mapComp f g [a]        = refl
mapComp f g (a :: as)  = cong (λ as → f (g a) :: as) (mapComp f g as)
```

Therefore, we can construct an element of type *Functor List* and clinch the proof that our *map* is a suitable choice:

```
FunctorList : Functor List
FunctorList = record {fmap    = map;
                      idLaw   = mapId;
                      compLaw = mapComp}
```

No problems with the polymorphism or higher-order nature of *map*, and, of course, no coverage problems. Motivated by this easy success, we proceed to formalize the vulnerability measure condition, starting first with the definition of increasing functions. We use the Agda standard library *IsPreorder* record for preorders, which is parameterized on the underlying equivalence (for which we use _≡_ throughout):

```
IsIncreasing : {A : Set} (_≤_ : A → A → Set) →
               (A → A) → Set
IsIncreasing (_≤_) f = ∀ a → a ≤ f a

VulnMeas : {F : Set → Set} → Functor F →
           {V : Set} → {_≤_ : V → V → Set}  → IsPreorder _≡_ _≤_ →
           {W : Set} → {_⊑_ : W → W → Set} → IsPreorder _≡_ _⊑_ →
           (m : F V → W) → Set
VulnMeas {F} fF {V} {_≤_} p≤ {W} {_⊑_} p⊑ m =
  (i : V → V) → IsIncreasing _≤_  i →
                IsIncreasing _⊑m_ (fmap i)
  where fmap = Functor.fmap fF
        _⊑m_ : F V → F V → Set
        x ⊑m y = m x ⊑ m y
```

This is a virtually literal translation of Definition 1, and not more trouble to write than the *testMonotonicity* function above.

The Agda versions of our vulnerability measure candidates are also cut & paste productions from the Haskell code, except for renamings due to the lack of type classes in Agda:

```
sumList :   List (Float × Int) → Float × Int
sumList  =  fold id f
  where f : Float × Int → Float × Int → Float × Int
        f (x, n) (x′, n′)  =  (x +f x′, n +i n′)
supList  :  List (Float × Int) → Float × Int
supList  =  fold id f
  where f : Float × Int → Float × Int → Float × Int
        f (x, n) (x′, n′)  =  (maxf x x′, maxi n n′)
```

In both cases, the arguments (*id* and f) that *fold* receives are monotonic functions, and it is easy to see that this is a sufficient condition for a vulnerability measure. Formulating this property in Agda raises no unexpected difficulties:

```
IsMonotonous : {A : Set} → {_⩽^A_ : A → A → Set} → (pA : IsPreorder _≡_ _⩽^A_) →
               {B : Set} → {_⩽^B_ : B → B → Set} → (pB : IsPreorder _≡_ _⩽^B_) →
               (A → B) →
               Set
IsMonotonous {A} {_⩽^A_} pA {B} {_⩽^B_} pB f  =
  (a1 a2 : A) → (a1 ⩽^A a2) → f a1 ⩽^B f a2
IsMonotonous2 : {A : Set} → {_⩽^A_ : A → A → Set} → (pA : IsPreorder _≡_ _⩽^A_) →
                {B : Set} → {_⩽^B_ : B → B → Set} → (pB : IsPreorder _≡_ _⩽^B_) →
                {C : Set} → {_⩽^C_ : C → C → Set} → (pC : IsPreorder _≡_ _⩽^C_) →
                (A → B → C) →
                Set
IsMonotonous2 {A} {_⩽^A_} pA {B} {_⩽^B_} pB {C} {_⩽^C_} pC f  =
  (a1 a2 : A) → (a1 ⩽^A a2)  →
  (b1 b2 : B) → (b1 ⩽^B b2)  → f a1 b1 ⩽^C f a2 b2
foldMeas : {A : Set} → {_⩽^A_ : A → A → Set} →   (pA : IsPreorder _≡_ _⩽^A_) →
           {B : Set} → {_⩽^B_ : B → B → Set} →   (pB : IsPreorder _≡_ _⩽^B_) →
           (w : A → B)         → IsMonotonous  pA pB      w →
           (c : A → B → B) → IsMonotonous2 pA pB pB c  →
           VulnMeas FunctorList pA pB (fold w c)
```

Folding monotonic functions over non-empty lists produces vulnerability measures: how hard is it to convince the type checker of this fact? Perhaps surprisingly, not hard at all. Agsy finds out all by itself that increasing the elements of a singleton list and applying a monotonic function to the result is going to result in an increased measure:

```
foldMeas pA pB w monw c mon2c i isInc [a]  =  monw a (i a) (isInc a)
```

More impressively, in the inductive case, after the gentle nudge to apply the monotonicity of the second argument to fold, Agsy can fill in all the arguments to mon_2c except for the last one, the induction hypothesis:

```
foldMeas pA pB w monw c mon2c i isInc (a :: as)  =
  mon2c a (i a) (isInc a)
        (fold w c as)
        (fold w c (fold (λ x → [i x]) (λ x → _::_ (i x)) as))
        ?
```

which we fill in and, after tempering a bit Agsy's eagerness to reduce every term to normal form, we reach the final version:

```
foldMeas pA pB w monw c mon2c i isInc (a :: as) =
  mon2c a              (i a)                  (isInc a)
        (fold w c as) (fold w c (map i as)) (foldMeas pA pB w monw c mon2c i isInc as)
```

All that remains to do in order to ensure that our candidates, *sumList* and *supList* are indeed vulnerability measures is to prove the monotonicity of $id, +_f, +_i, maxf, maxi$. Well, we cannot! *Float* and *Int* are machine built-in types, which Agda allows us access with a bit of builtin-trickery:

```
postulate Float : Set {-# BUILTIN FLOAT Float #-}
primitive
  primFloatPlus : Float → Float → Float
  primFloatLess : Float → Float → Bool
_+_ : Float → Float → Float
_+_ = primFloatPlus
_⩽_ : Float → Float → Bool
_⩽_ = primFloatLessThan
```

And the same thing again for *Int*. But, beyond the signature of these functions, the type checker knows nothing about them, and any additional property must be postulated, for example:

```
postulate ⩽fRefl  : (x : Float) → x ⩽f x
postulate ⩽fTrans : (x y z : Float) →
                    x ⩽f y → y ⩽f z → x ⩽f z
postulate +fmon   : (x y x′ y′ : Float) →
                    x ⩽f x′ → y ⩽f y′ →
                    (x +f y) ⩽f (x′ +f y′)
```

where $\leqslant f$ is a suitably lifted representation of the primitive boolean relation. The type checker accepts then (but does not guarantee) that these properties hold, and we obtain thus a conditional proof of correctness, with the implementational aspects nicely tucked away and signalled by the **postulate** keyword.

Alternatively, we can use Peano naturals instead of *Int* and rationals instead of *Float*, for which we *can* prove the required properties, and obtain an unconditional result (and a less efficient program). Eventually, one expects such properties to be part of standard libraries, and have an even easier time switching from one datatype to another. In any case, the most difficult part of the job, proving that a fold gives a vulnerability measure, is independent of the specific datatype considered.

To sum up, formulating the vulnerability measure condition via the Curry-Howard isomorphism is not more difficult than coming up with the corresponding tests, while proving it for the cases we considered is easier and more general than implementing those tests. The conceptual and implementational aspects are cleanly separated, and the problematic spots highlighted by the **postulate** keyword.

5 Conclusions

There have been several papers lately that show the advantages of dependently-typed programming languages for embedded domain-specific languages [5, 19, 24], and we have just provided another example.

A feature that distinguishes our work from the others is that it brings us in contact with scientific programming: the kind of programming that covers the models used to generate the possible trajectories to be measured. The scientific programming community often tackles problems with the sort of features our example illustrates, where exhaustive testing is not feasible and formal proofs of correctness might be easier. Scientific programmers tend also to be familiar with mathematical proof in an informal context: many numerical methods are justified by some sort of informal proof of correctness, which is then a candidate for translating to a formal context. The question therefore is, why is formal proof not used more frequently in scientific programming?

One reason is probably that usable implementations of dependently-typed programming languages have not been around very long. Moreover, the experience we have accumulated with them has been more on the discrete, algebraic side and rather less on the continuous, real analysis side which is important for scientific programming. The Agda standard library [26], young as it is (currently at version *0.6*), implements many kinds of algebraic structures, but has no mention of the *Float* datatype or real numbers. There are, to our knowledge, no dependently-typed libraries available for doing the sort of things that a scientific programmer takes for granted: solving linear systems, factorizing matrices, interpolating real functions, optimization, and so on.

Developing such libraries in a dependently-typed programming language is quite challenging. Consider, for example, that in order to implement an optimization method, one has to specify exactly what is meant by "optimization": does the method return the exact solution or just an approximation of it?

We can attempt to obtain the exact solution if we work with constructive real numbers in the realm of constructive real analysis, as suggested, for example, by Bishop [3]. There are several representations of exact real numbers: the ones most used in constructive numerical analysis are based on the work of Russell O'Connor in Nijmegen [22, 23]. Validated numerical methods via constructive analysis is still a research subject. There are promising results [12], but they are quite far from providing a usable basis for scientific programming. In particular, there are no library functions available yet for solving a linear system of equations.

An alternative approach is to content ourselves with an approximate solution. After all, the vast majority of numerical libraries available today work with floating point numbers and thus abandon the search for an exact solution from the beginning. Here the challenge is to specify what *is* being computed: what guarantees are made about the quality of the approximation delivered? Existing libraries tend to be surprisingly vague here, encouraging a trial-and-error approach and relying on the expertise of the user. The arguments for why a certain method should lead to a good approximation of the solution are also often expressed in terms of exact real numbers and therefore can only be formalized with the help of postulates, as we have done above.

To do better, one has to formalize the properties of floating-point numbers as expressed in the IEEE 754 or 854 floating-point arithmetic standard. Several such formalizations have been achieved in PVS [18], HOL [10], and Coq [4], and have been used to verify the implementation of algorithms for fundamental and relatively simple functions, such as the square root or the exponential. To our knowledge, no substantial numerical methods have

yet been verified. Moreover, this kind of work is hard to do in an academic context, and we might have to wait until industry is motivated enough to fund it.

Until such a time, the best that we can do is to separate the problems that require the continuous / analytic from those that deal more with the discrete / algebraic, and prove the correctness of the latter conditional on (postulated) correctness of the former, which we can at most test. In this sense, in the above examples, we were indeed lucky, having to deal only with algebraic structures such as preorders and lists, and being satisfied with correctness conditioned on the field structure of floating-point numbers and integers (a structure they, in fact, do not have!).

Acknowledgement

The authors gratefully acknowledges the fruitful discussions with Andreas Abel, Jean-Philippe Bernardy, Paul Flondor, and the members of the Cartesian Seminar at PIK.

References

1 P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

2 Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In Andrew Gordon, editor, *European Symposium on Programming*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010.

3 E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

4 S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011. IEEE CS Press.

5 Edwin C. Brady. Idris — systems programming meets full dependent types. In *Proc. 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54. ACM, 2011.

6 T. Cannon, J. Twigg, and J. Rowell. Social vulnerability, sustainable livelihoods and disasters. Technical report, AON Benfield UCL Hazard Research Center, 2002. Available at `http://www.abuhrc.org/Documents/Social_vulnerability_sust_live.pdf`.

7 J. Chakraborty, G.A. Tobin, and B.E. Montz. Population evacuation: assessing spatial variability in geophysical risk and social vulnerability to natural hazards. *Natural Hazards Review*, pages 22–33, February 2005.

8 K. Claessen and J. Hughes. Specification based testing with QuickCheck. In *The Fun of Programming*, Cornerstones of Computing, pages 17–40. Palgrave, 2003.

9 T. Feldbrügge and J. von Braun. Is the world becoming a more risky place? Trends in disasters and vulnerability to them. Technical Report 46, Center for Development Research, Bonn, 2002.

10 John Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997. Available as `http://www.cl.cam.ac.uk/~jrh13/papers/tang.html`.

11 C. Ionescu. *Vulnerability Modeling and Monadic Dynamical Systems*. PhD thesis, Department of Mathematics and Informatics, Free University Berlin, February 2009.

12 R. Krebbers and B. Spitters. Computer certified efficient exact reals in Coq. In James H. Davenport et al., editors, *Calculemus/MKM*, volume 6824 of *LNCS*, pages 90–106. Springer, 2011.

13 D. Lemmen and F. Warren, editors. *Climate Change Impacts and Adaptation: A Canadian Perspective*. Natural Resources Canada, 2004.

14 Daniel Lincke, Patrik Jansson, Marcin Zalewski, and Cezar Ionescu. Generic libraries in C++ with concepts from high-level domain descriptions in Haskell: A DSL for computational vulnerability assessment. In Walid Taha, editor, *IFIP Working Conf. on Domain Specific Languages*, volume 5658 of *LNCS*, pages 236–261, 2009.

15 F. Lindblad and M. Benke. A tool for automated theorem proving in Agda. In Jean-Christophe Filliâtre et al., editors, *Types for Proofs and Programs, International Workshop, TYPES 2004*, LNCS, pages 154–169. Springer, 2006.

16 J.J. McCarthy, O. Canziani, N.A. Leary, D.J. Dokken, and K.S. White, editors. *Climate Change 2001: Impacts, Adaptation and Vulnerability. Contribution of Working Group II to the Third Assessment Report of the Intergovernmental Panel on Climate Change.* Cambridge University Press, 2001.

17 M.J. Metzger and D. Schröter. Towards a spatially explicit and quantitative vulnerability assessment of environmental change in Europe. *Regional Environmental Change*, 6(4):201–216, 2006.

18 Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical report, 1995. Technical Memorandum 110167, NASA, Langley Research. Availble as `http://nasa1995.tpub.com/NASA-95-tm110167/`.

19 Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *Proc. 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 169–180. ACM, 2010.

20 G.J. Myers. *The Art of Software Testing. Second edition, revised and updated by T. Badgett and T.M. Thomas with C. Sandler.* John Wiley & Sons, Inc., 2004.

21 U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD thesis, Chalmers University of Technology, September 2007.

22 R. O'Connor. A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 1:129–159, 2007.

23 Russell O'Connor. Certified exact transcendental real number computation in Coq. In Otmane Ait Mohamed et al., editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 246–261. Springer, 2008.

24 N. Oury and W. Swierstra. The power of Pi. In *International Conference on Functional Programming*, pages 39–50. ACM, 2008.

25 B. Pascal. *Minor Works, translated by O. W. Wright*, volume XLVIII, Part 2 of *The Harvard classics*. P.F. Collier & Son, 1909-14.

26 The Agda Team. The Agda Wiki, 2011. Available from `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Includes documentation, links to the Agda implementation and to the standard library.

27 K. Thywissen. Components of risk, a comparative glossary. *SOURCE - Studies Of the University: Research, Counsel, Education*, 2, 2006.

28 UN/ISDR (United Nations International Strategy for Disaster Reduction). *Living with Risk. A Global Review of Disaster Reduction Initiatives.* United Nations, Geneva, 2004.

Verification of redecoration for infinite triangular matrices using coinduction

Ralph Matthes[1] and Celia Picard[2]

1 **Institut de Recherche en Informatique de Toulouse (IRIT), C.N.R.S. and University of Toulouse, France**
2 **Institut de Recherche en Informatique de Toulouse (IRIT), University of Toulouse, France**

Abstract

Finite triangular matrices with a dedicated type for the diagonal elements can be profitably represented by a nested data type, i. e., a heterogeneous family of inductive data types, while infinite triangular matrices form an example of a nested coinductive type, which is a heterogeneous family of coinductive data types.

Redecoration for infinite triangular matrices is taken up from previous work involving the first author, and it is shown that redecoration forms a comonad with respect to bisimilarity.

The main result, however, is a validation of the original algorithm against a model based on infinite streams of infinite streams. The two formulations are even provably equivalent, and the second is identified as a special instance of the generic cobind operation resulting from the well-known comultiplication operation on streams that creates the stream of successive tails of a given stream. Thus, perhaps surprisingly, the verification of redecoration is easier for infinite triangular matrices than for their finite counterpart.

All the results have been obtained and are fully formalized in the current version of the Coq theorem proving environment where these coinductive datatypes are fully supported since the version 8.1, released in 2007. Nonetheless, instead of displaying the Coq development, we have chosen to write the paper in standard mathematical and type-theoretic language. Thus, it should be accessible without any specific knowledge about Coq.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases nested datatype, coinduction, theorem proving, Coq

Digital Object Identifier 10.4230/LIPIcs.TYPES.2011.55

1 Introduction

Redecoration for the finite triangles has been verified against a list-based model in previous work [10]. This is the point of departure for our present paper.

Finite triangles can be represented by "triangular matrices", i. e., finite square matrices, where the part below the diagonal has been cut off. Equivalently, one may see them as symmetric matrices where the redundant information below the diagonal has been omitted. The elements on the diagonal play a different role than the other elements in many mathematical applications, e. g., one might require that the diagonal elements are invertible (non-zero). This is modeled as follows: a type E of elements outside the diagonal is fixed throughout (we won't mention it as parameter of any of our definitions), and there is a type of diagonal elements that enters all definitions as an explicit parameter. More technically, if A is the type of diagonal elements, then $Tri_{fin}\, A$ shall denote the type of finite triangular matrices with A's on the diagonal and E's outside (see Figure 1). Then, Tri_{fin} becomes a family of

© R. Matthes and C. Picard;
licensed under Creative Commons License BY-ND
18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson, Bengt Nordström; pp. 55–69

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

types, indexed over all types, hence a type transformation. Moreover, the different $\mathit{Tri}_{\mathit{fin}}\, A$ are inductive datatypes that are all defined simultaneously, hence they are an "inductive family of types" or "*nested datatype*" [4].

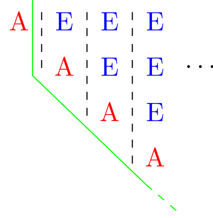


Figure 1 Dividing a triangle into columns

If we cut the triangle into the first column and the rest, we get one element of A and a "*trapezium*", with an uppermost row solely consisting of E's. In order not to have to ensure explicitly by a dependent type that the number of columns is coherent, the solution is to transform the trapezium into a triangle, integrating the side diagonal (just above the diagonal) into the diagonal itself, as shown in Figure 2. From the left to the right, the lowermost element of E in each column is paired with the element of A on the diagonal, and the other elements of E remain untouched.

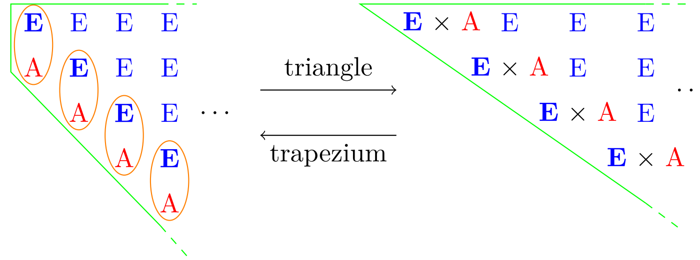


Figure 2

Following this remark, the triangles can be defined theoretically [1], and in Coq and Isabelle by the following constructors [10]:

$$\frac{a : A}{\mathit{sg}_{\mathit{fin}}\, a : \mathit{Tri}_{\mathit{fin}}\, A} \qquad\qquad \frac{a : A \qquad t : \mathit{Tri}_{\mathit{fin}}(E \times A)}{\mathit{constr}_{\mathit{fin}}\, a\, t : \mathit{Tri}_{\mathit{fin}}\, A}$$

▶ Remark. In this paper, single-lined inference rules denote inductive definitions, double-lined inference rules are for coinductive definitions.

In more theoretical terms, $\mathit{Tri}_{\mathit{fin}}$ is modeled as the *least* solution to the fixed-point equation

$$\mathit{Tri}_{\mathit{fin}}\, A = A + A \times \mathit{Tri}_{\mathit{fin}}\, (E \times A)$$

The left summand corresponds to a triangle that only consists of a single element of A (a singleton), thus ensuring the base case.

The algorithm of redecoration (see work by Uustalu and Vene for the general categorical notion [14]) is the following: for a given redecoration rule $f : \mathit{Tri}_{\mathit{fin}}\, A \to B$, it is a function $\mathit{redec}\, f$ that redecorates A-triangles t (elements of $\mathit{Tri}_{\mathit{fin}}\, A$) into B-triangles by applying f to the whole triangle t to obtain the new top element, and then by successively applying the same operation to the triangle cut out from the remaining trapezium. This ends in the singleton case where f is applied to it and the result is turned into a triangle by applying $\mathit{sg}_{\mathit{fin}}$. This algorithm only changes the diagonal elements in A into elements of B, as shown in Figure 3. We do not give the formal definition of redecoration for finite triangles here.

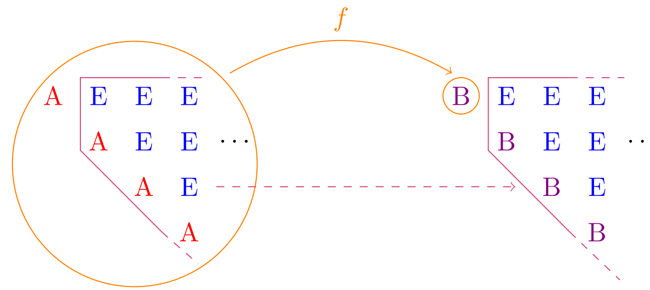


Figure 3 Redecoration

The redecoration for infinite triangles [1] has not yet been verified. This is what we intend to do in this paper.

Reasoning about nested coinductive types naturally rests on observational equality, just as for ordinary coinductive types, and since version 8.2, Coq greatly helps in using the rewrite mechanism for Leibniz equality also for the notion of bisimilarity of infinite triangular matrices. With respect to that notion of equality, redecoration is shown to form a (sligthly weakened form of) comonad, and its implementation is compared with an alternative one based on streams of streams.

These new results come with a full formalization in Coq [9], and limitations of what Coq recognizes as a guarded definition make the theoretical development more challenging, but we still obtained smooth results without an excessive overhead that would be imposed by a naive dualization of the formalization for the finite triangles [10].

In Section 2, inspired by the previous theoretical development [1], we introduce the dual to the definition of finite triangles [10]. We present it with all the tools necessary to define redecoration. We then propose a definition for the redecoration algorithm on these infinite triangles and add further tools and properties. In Section 3, we change the point of view in the observation of the triangles. We give an alternative definition for the infinite triangles, considering this new approach, and provide various tools. We also show that this new representation is equivalent to the previous one. Finally, we propose two ways of defining redecoration, trying always to simplify and generalize our definitions and show their adequation with previous definitions.

Since the results are fully formalized in the current version of Coq, hence ensuring complete and sound proofs, we took the liberty to write the paper in standard mathematical and type-theoretic language and also to omit most proofs. Therefore, it should be accessible without any specific knowledge about Coq. For the study of the development [9], the Coq'Art book [3] should mostly suffice, but the (type) class mechanism [12] and the revised setoid rewriting mechanism based on it have to be consulted elsewhere – by default in the Coq Reference Manual [13].

2 Reference Representation with a Coinductive Family

Dually to the representation of finite triangles discussed in the introduction, "triangular matrices" are now introduced as *infinite* square matrices, where the part below the diagonal has been cut off. Recall that a type E of elements outside the diagonal is fixed throughout. If A is the type of diagonal elements, then $\mathit{Tri}\,A$ shall denote the type of infinite triangular matrices with A's on the diagonal and E's outside. The different $\mathit{Tri}\,A$ are coinductive datatypes that are all defined simultaneously, as was the case for $\mathit{Tri}_{\mathit{fin}}$, hence they are a "coinductive family of types" or "*nested codatatype*", as will be developed below.

2.1 Infinite triangles as nested coinductive type

Infinite triangles can also be visualized as in Figure 1, this time with the dots representing an infinite extension.

If we now cut the triangle into the first column and the rest, we get one element of A (as before for Tri_{fin}) and a trapezium, with an uppermost row consisting of infinitely many E's.

The n-th column consists of an element of A on the diagonal and n elements of E above the diagonal, as in the case of Tri_{fin}. As before, we do not want to parameterize the type of the columns by their index and instead integrate the side diagonal into the diagonal – and this has to be done corecursively [1]. This integration is possible since trapeziums are again in one-to-one correspondence to triangles, as shown in Figure 2, now interpreted infinitely. In this figure, the trapezium to the left is considered as the "trapezium view" of the triangle to the right. Vice versa, the triangle to the right is the "triangle view" of the trapezium to the left.

We now formalize triangles through the following constructor that has to be interpreted coinductively.

▶ **Definition 1** (Tri, defined coinductively)**.**

$$\frac{a : A \qquad t : Tri(E \times A)}{constr\, a\, t : Tri\, A}$$

with A a type variable.

This means that the types $Tri\, A$ for all types A are simultaneously conceived as greatest solution to the fixed-point equation

$$Tri\, A = A \times Tri(E \times A),$$

and $constr$ has two arguments instead of a pair of type $A \times Tri(E \times A)$ just for technical convenience.

The second argument to $constr$ corresponds to the triangle view of the trapezium in our visualization in Figure 1, but there is no passage between a trapezium and a triangle – this is only the motivation. In the formalization, there are only infinite triangles, but we set $Trap\, A := Tri(E \times A)$ to hint to the trapezium view of these triangles.

▶ **Definition 2** (Projections)**.**

$$\begin{array}{ll} top : \forall A.\ Tri\, A \to A & rest : \forall A.\ Tri\, A \to Trap\, A \\ top\,(constr\, a\, r) := a & rest\,(constr\, a\, r) := r \end{array}$$

This definition by pattern matching implicitly uses the direction from right to left in the above fixed-point equation. Thus, the top element and the trapezium part of a triangle are calculated by unfolding the fixed point.

In order to obtain the triangle that arises by cutting off the top row of a trapezium, we have to go through all the columns.

▶ **Definition 3** ($cut : \forall A. Trap\, A \to Tri\, A$, defined corecursively)**.**

$$cut\,(constr\,\langle e, a\rangle\, r) := constr\, a\,(cut\, r)$$

The definition does pattern matching on elements of $Trap\, A$ and constructs an element of the coinductive type $Tri\, A$. The subterm $cut\, r$ represents a corecursive call to cut, which is accepted also by the Coq system as admissible corecursion since it is placed directly as an argument to the constructor $constr$.

2.2 Redecoration for infinite triangles

We are heading for a corecursive definition of the generic redecoration operation *redec* on triangles. It has type

$$\forall A \forall B. \, (\mathit{Tri}\, A \to B) \to \mathit{Tri}\, A \to \mathit{Tri}\, B,$$

which is the type of a coextension operation for *Tri* viewed as support of a comonad. Coextension – also called cobind – is the dual of the extension / bind operation of a monad, which is so successfully used in the functional programming language Haskell. The counit for the comonad we are about to construct is our *top* operation.

Redecoration for *Tri* follows the same pattern as for $\mathit{Tri}_{\mathit{fin}}$, but the successive applications of the same operation will never reach a base case, as there is none in *Tri*.

Formally, this is done by a corecursive definition, where $\mathit{redec}\, f\, t$ for t of type $\mathit{Tri}\, A$ has to call itself with second argument of type $\mathit{Trap}\, A$, hence f is not even type-correct as a first argument in that corecursive call. Instead of $f : \mathit{Tri}\, A \to B$, a "lifted" version of f is needed that has type $\mathit{Tri}(E \times A) \to E \times B$.

▶ **Definition 4** ($\mathit{lift} : \forall A \forall B. \, (\mathit{Tri}\, A \to B) \to \mathit{Tri}(E \times A) \to E \times B$)**.**

$$\mathit{lift}\, f\, r := \langle \mathit{fst}(\mathit{top}\, r), f(\mathit{cut}\, r) \rangle \ \ ,$$

where *fst* is the first projection (from a pair to its first component).

The definition is illustrated in Figure 4.

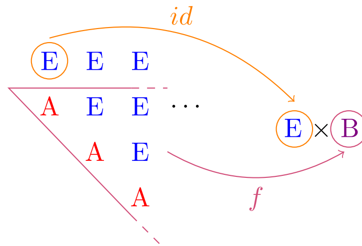


Figure 4 Definition of lifting

The formal definition of redecoration is as follows:

▶ **Definition 5** ($\mathit{redec} : \forall A \forall B. \, (\mathit{Tri}\, A \to B) \to \mathit{Tri}\, A \to \mathit{Tri}\, B$, defined corecursively)**.**

$$\mathit{redec}\, f\, t := \mathit{constr}\, (f\, t)\, \big(\mathit{redec}\, (\mathit{lift}\, f)\, (\mathit{rest}\, t)\big) \ \ ,$$

see Figure 5. This definition is accepted since it is guarded: the corecursive call to *redec* is as second argument to *constr*, and it does not matter that the argument f becomes $\mathit{lift}\, f$ there. A function argument that becomes more complicated in the recursive call is typical of recursion on nested datatypes, see, e. g., [1].

This completes the definition, but leaves open the question if this is really (in what sense) the cobind of a comonad and if it corresponds to operations that are easier to understand than corecursion on nested codatatypes. We note that recursion schemes for nested datatypes have been subject of a long line of research, starting from work by Bird and colleagues [4, 5].

2.3 Properties of redecoration

It is well-known that propositional equality $=$ (called Leibniz equality in Coq since $t_1 = t_2$ allows the replacement of t_1 by t_2 in any mathematical context) cannot suffice as criterion for

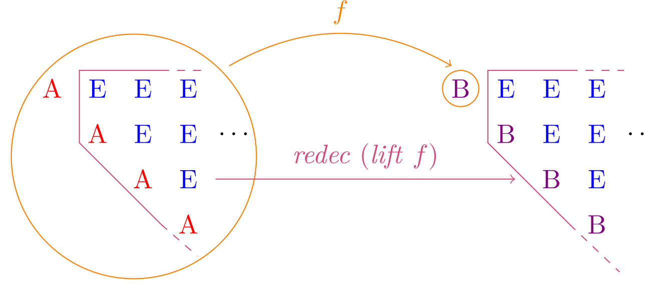


Figure 5 Definition of redecoration

the correctness of elements that are calculated in coinductive types. Propositional equality cannot be established by coinductive reasoning because this is confined to coinductively defined conclusions, and propositional equality is not coinductive (in Coq, it is defined inductively). We write $Rel\, C$ for the type of the binary relations on C, and we use these relations in infix notation. In Coq, their type is $C \to C \to Prop$, where $Prop$ is the universe of propositions.

▶ **Definition 6** ($\simeq : \forall A.\, Rel(Tri\, A)$, defined coinductively).

$$\frac{t_1\,,\, t_2 : Tri\, A \qquad top\, t_1 = top\, t_2 \qquad rest\, t_1 \simeq rest\, t_2}{t_1 \simeq t_2}$$

It is easy to show that $\simeq$ is an equivalence relation for any argument type A. It is an equivalence relation but not a congruence: for every operation of interest we have to establish compatibility with bisimilarity. This is in particular easily done for the projection functions *top* and *rest* and for the *cut* operation.

Using this notion of bisimilarity, we can show that *redec* is extensional in its function argument (modulo $\simeq$), using full extensionality of *lift*:

▶ **Lemma 7.** $\forall A \forall B \forall (f\, f' : Tri\, A \to B).\, (\forall t, f\, t = f'\, t) \Rightarrow \forall t, lift\, f\, t = lift\, f'\, t$

▶ **Lemma 8.** $\forall A \forall B \forall (f\, f' : Tri\, A \to B).\, (\forall t, f\, t = f'\, t) \Rightarrow \forall t, redec\, f\, t \simeq redec\, f'\, t$

The main properties of *redec* we are interested in express that *top* and *redec* together constitute a comonad for “functor” *Tri*. The precise categorical definition in coextension form (with a cobind operation instead of the traditional comultiplication) is, e. g., given in [14]. Here, we give the constructive notion we use in this paper, and it is parameterized by an equivalence relation while classically, only mathematical equality $=$ is employed.

▶ **Definition 9** (Constructive comonad). A constructive comonad consists of a type transformation T, a function $counit : \forall A.\, T\, A \to A$, a function $cobind : \forall A \forall B.(T\, A \to B) \to T\, A \to T\, B$ and an equivalence relation $\approxeq : \forall A.\, Rel(T\, A)$ such that the following comonad laws hold:

$$\forall A \forall B \forall f^{T\, A \to B} \forall t^{T\, A}.\, counit(cobind\, f\, t) = f\, t \tag{1}$$

$$\forall A \forall t^{T\, A}.\, cobind\, counit_A\, t \approxeq t \tag{2}$$

$$\forall A \forall B \forall f^{T\, A \to B} \forall g^{T\, B \to C} \forall t^{T\, A}.\, cobind\, (g \circ cobind\, f)\, t \approxeq cobind\, g\, (cobind\, f\, t) \tag{3}$$

Here, in order to save space, we gave the type information for the term variables as superscripts. The index A to *counit* is meant to say that the type parameter to *counit* is set to A – in all other cases, we leave type instantiation implicit.

▶ **Definition 10** (Constructive weak comonad)**.** A constructive weak comonad is defined as a constructive comonad, but where the equation in (3) is restricted to functions g that are compatible with $\approxeq$ in the following sense: $\forall t\,t', t \approxeq t' \Rightarrow g\,t = g\,t'$.

▶ **Lemma 11.** *The type transformation Tri, the projection function top and redec form a constructive weak comonad with respect to* $\simeq$.

The first comonad law is satisfied in an especially strong form: $top(redec\,f\,t)$ actually *is* $f\,t$ by definition. The other comonad laws go through with suitable generalizations of the lemmas – in order to ensure guardedess of the proofs. The current solution is unspectacular, but it was not obvious how to do it (much more complicated solutions were found on the way and are now obsolete). We only show the strengthening of the second comonad law, but it is the same style for the third one.

▶ **Lemma 12** (strengthened form of second comonad law for *redec*)**.**

$$\forall A \forall (f : Tri\,A \to A).\,(\forall (t : Tri\,A), f\,t = top\,t) \Rightarrow \forall (t : Tri\,A).\,redec\,f\,t \simeq t$$

The proof is by coinduction and uses Lemma 7. Obviously, this implies the second comonad law. For all the details, see our formalization in Coq [9]. We only get a weak comonad because proving pointwise equality of $lift(g \circ (redec\,f))$ and $(lift\,g) \circ (redec(lift\,f))$ requires compatibility of g with $\simeq$, and this is a crucial step for proving the third comonad law.

When defining the *cut* operation, one might naturally want to get also the part that has been cut out (the elements of E). These elements are given by the following function:

▶ **Definition 13** ($es_cut : \forall A.\,Trap\,A \to Str\,E$, defined corecursively)**.**

$$es_cut\,(constr\,\langle e, a\rangle\,r) := e :: (es_cut\,r)$$

▶ Remark. In the standard library of Coq, the type of streams with elements in type C are predefined, and we can represent this definition as follows:

$$\frac{c : C \qquad s : Str\,C}{c :: s : Str\,C}$$

The projection functions are called hd and tl. They are such that $hd(c :: s) = c$ and $tl(c :: s) = s$. We will also use the map function defined by $map\,f\,(c :: s) = f\,c :: (map\,f\,s)$.

Using es_cut, we can define the first row of E elements in a triangle as

$$frow : \forall A.\,Tri\,A \to Str\,E \qquad frow\,t := es_cut\,(rest\,t)$$

Once we have these definitions, we might want to be able to "glue" the two cut parts in order to recreate the original trapezium. This is done by the function *addes*

▶ **Definition 14** ($addes : \forall A.\,Str\,E \to Tri\,A \to Trap\,A$, defined corecursively)**.**

$$addes\,(e :: es)\,(constr\,a\,r) := constr\,\langle e, a\rangle\,(addes\,es\,r)$$

And it is then easy to show that *addes* indeed performs the gluing:

$$\forall A \forall (r : Trap\,A).\,addes\,(es_cut\,r)\,(cut\,r) \simeq r$$

3 Another Conception of Triangles

In this section, we show another way to perceive and represent infinite triangles. And we propose two ways of defining redecoration on this new representation.

3.1 A new definition using streams

In the previous section, we always visualized the infinite triangles by their columns. Indeed, we said that a triangle was a first column with only one element of type A (the element of the diagonal) and a trapezium, itself actually a triangle, as suggested in Figure 1.

While elements of the finite triangles in $\mathit{Tri}_{\mathit{fin}}\,A$ (see Section 1) are (globally) finite, also all the columns of our infinite triangles in $\mathit{Tri}\,A$ are finite. In the work that we started from for this article [10], redecoration on $\mathit{Tri}_{\mathit{fin}}$ is verified against a model where triangles are represented by finite lists of columns, where each column consists of the diagonal element in A and a finite list of elements in E. A naive dualization of that approach would consist in taking as representation of infinite triangles streams of columns that would be formed as for the finite ones. This mixture of inductive and coinductive datatypes is notoriously difficult to handle. We have been confronted with this problem many times in the last few years, as can be seen in the second author's thesis [11] which deals with this kind of problems particularly in Coq. But in other proof assistants, the same kind of issues has appeared; there is also an experimental solution in Agda [6, 7]. Still, the representation of infinite triangles mixing inductive and coinductive datatypes can be carried out, but we refrain from presenting this column-based approach here.

However, we can also visualize triangles the other way around. We now consider the triangle by its rows, as suggested in Figure 6. Then, on any row, we have one element of

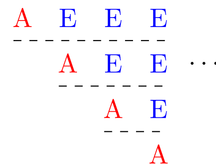


Figure 6 Dividing a triangle into rows

type A and infinitely many elements of type E. And we also have infinitely many rows. Here, nothing is finite (only the single element of A at the head of each row, but this is not a problem), therefore, we do not have any embedded inductive type in our description – unlike in the columnwise decomposition mentioned above. This new visualization can be represented as a stream of pairs made of one element of type A and a stream of elements of type E.

▶ **Definition 15.** $\mathit{Tri}'\,A := \mathit{Str}(A \times \mathit{Str}\,E)$

Actually, following the definition of Str, we can read this new definition of the triangles as consisting of three parts: the top element, the stream of elements of E of the first row and the triangle corresponding to the rest, as shown in Figure 7.

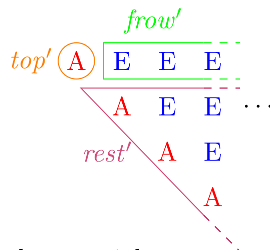


Figure 7 Conceptualizing a triangle as a triple

We define functions that allow us to access to each of these elements:

▶ **Definition 16** (Projections).

$$top' : \forall A.\, Tri'\, A \to A \qquad frow' : \forall A.\, Tri'\, A \to Str\, E \qquad rest' : \forall A.\, Tri'\, A \to Tri'\, A$$
$$top'\, (\langle a, es\rangle :: t) := a \qquad frow'\, (\langle a, es\rangle :: t) := es \qquad rest'\, (\langle a, es\rangle :: t) := t$$

Notice that *rest* and *rest′* are conceptually different – the former yields the trapeziums after cutting off the first column, the latter triangles after cutting off the first row.

To compare two elements of Tri', we need a notion of bisimilarity, which on Str is pre-defined in Coq as follows:

▶ **Definition 17** ($\equiv : \forall C.\, Rel(Str\, C)$, defined coinductively).

$$\frac{s_1\,,\, s_2 : Str\, C \qquad hd\, s_1 = hd\, s_2 \qquad tl\, s_1 \equiv tl\, s_2}{s_1 \equiv s_2}$$

However, we cannot use it directly. Indeed, we would need to prove, for two triangles t_1 and t_2 that their first rows are Leibniz-equal, i. e., $frow'\, t_1 = frow'\, t_2$. This is too strict, since the rows are defined partially coinductively (because of the stream of E's). Therefore, we need to define a new relation on Tri' that will compare the three elements of the triangles. The tops can be compared through Leibniz equality, the first rows can be compared using $\equiv$ and the rests with the relation on Tri', corecursively.

▶ **Definition 18** ($\cong : \forall A.\, Rel(Tri'\, A)$, defined coinductively).

$$\frac{t_1\,,\, t_2 : Tri'\, A \qquad top'\, t_1 = top'\, t_2 \qquad frow'\, t_1 \equiv frow'\, t_2 \qquad rest'\, t_1 \cong rest'\, t_2}{t_1 \cong t_2}$$

It is immediate to show that $\cong$ is an equivalence relation.

In order to validate this view of the triangles, we want to show that it is indeed equivalent to the original one. Therefore, we are going to show that there is a bijection between the two definitions (modulo pointwise bisimilarity). To do so we define two conversion functions (*toStreamRep*, from Tri to Tri' and *fromStreamRep* for the other way around) and show that their compositions are pointwise bisimilar to the identity.

The two conversion functions are quite natural. To transform an element of $Tri\, A$ into an element of $Tri'\, A$, we need to reconstruct from the original triangle the three elements of $Tri'\, A$. The top remains the original top, this is trivial. The first row of elements of E is given by *frow*. Finally, the triangle has to be transformed again by *toStreamRep* from the rest of the triangle with the first row cut out by the function *cut*. The calculation for the different parts is represented in Figure 8.

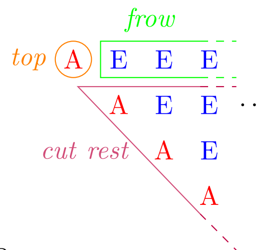


Figure 8 Definition of *toStreamRep*

▶ **Definition 19** ($toStreamRep : \forall A.\, Tri\, A \to Tri'\, A$, defined corecursively).

$$toStreamRep\, t := \langle top\, t, frow\, t\rangle :: toStreamRep\,(cut\,(rest\, t))$$

The definition of *fromStreamRep* is also quite intuitive. We have to construct the two elements that compose elements of type *Tri*. The top remains the top as before, this is again trivial. For the rest, we have to "glue" the first row to the rest of the triangle (basically the inverse of the *cut* and *es_cut* functions on *Tri′*) before transforming it again. We call *addes′* the function that performs this operation, as shown in Figure 9.

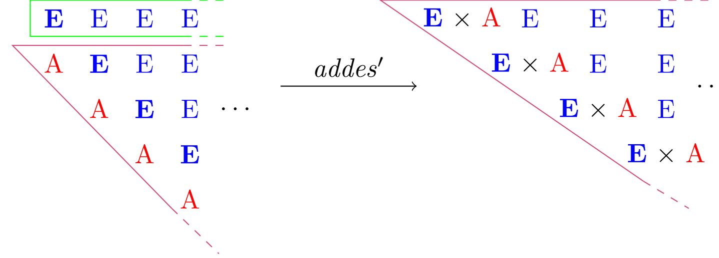


Figure 9 Definition of *addes′*

▶ **Definition 20** ($addes' : \forall A.\, Str\, E \to Tri'\, A \to Tri'(E \times A)$, defined corecursively).

$$addes'\,(e :: es)\,t := \langle\langle e, top'\,t\rangle, es\rangle :: addes'\,(frow'\,t)\,(rest'\,t)$$

▶ **Definition 21** ($fromStreamRep : \forall A.\, Tri'\, A \to Tri\, A$, defined corecursively).

$$fromStreamRep\, t := constr\,(top'\,t)\,\big(fromStreamRep\,(addes'\,(frow'\,t)\,(rest'\,t))\big)$$

▶ Remark. Our first idea was to do the gluing after the transformation. Indeed, as the transformation does not affect the elements of E, it seemed more natural to us not to submit this part to the corecursive call of the transformation. Thus, we wanted to define *fromStreamRep* coinductively as follows:

$$fromStreamRep\, t := constr\,(top'\,t)\,(addes\,(frow'\,t)\,(fromStreamRep\,(rest'\,t)))$$

However, even if this seems harmless, this definition cannot be accepted by Coq since the corecursive call to *fromStreamRep* is not guarded (it is an argument of *addes* and not of a constructor). Nevertheless, we have shown that the solution to the previous equation is unique with respect to pointwise bisimilarity and that *fromStreamRep* of Definition 21 satisfies it.

▶ **Lemma 22.** $\forall A \forall (t : Tri'\, A).\, toStreamRep\,(fromStreamRep\, t) \cong t$

Proof. To prove this result, we actually prove the following stronger result that we then only instantiate to finish the proof:

$$\forall A \forall (t : Tri'\, A)(u : Tri\, A),\ toStreamRep\,(fromStreamRep\, t) \cong u \Rightarrow t \cong u$$

The proof of this statement is a simple coinduction, that uses some straightforward results on *cut* and *addes′*. ◀

▶ **Lemma 23.** $\forall A \forall (t : Tri\, A).\, fromStreamRep\,(toStreamRep\, t) \simeq t$

Proof. We use the same technique as before. We prove a stronger result that we instantiate to prove our lemma:

$$\forall A \forall (t : Tri\, A)(u : Tri'\, A).\, fromStreamRep\,(toStreamRep\, t) \simeq u \Rightarrow t \simeq u$$

Here again, the proof is a straightforward coinduction using compatibility of *top* and *rest* with $\simeq$ and a simple result on *addes′*. ◀

3.2 Redecoration on Tri'

Thus, we have a completely different view of the triangles, but still, it is fully equivalent to the original one. The interest of this view is that now the redecoration is very easy to perform. Indeed, before, the tricky part was that we had to lift the function f to trapeziums, and therefore to cut out the elements of E remaining (implicitly) from the first row. The problem was that we roughly had to cut out a row, while we were reasoning on columns. Here, as we directly reason on rows, it is much easier. As shown in Figure 10, the three elements of the transformed triangle will be:

- the top is the application of f to the whole triangle (as before)
- the first row of elements of E is the same row as in the original triangle (and as we said we have direct access to it)
- the rest of the triangle is the application of the redecoration function to the rest

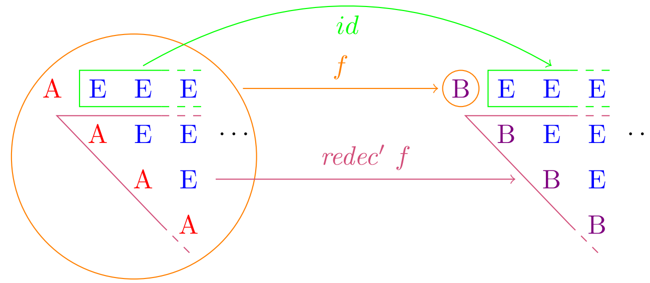


Figure 10 Definition of redecoration

Therefore, we can define the redecoration function for Tri' as follows:

▶ **Definition 24** ($\mathit{redec}' : \forall A \forall B. (\mathit{Tri}'\ A \to B) \to \mathit{Tri}'\ A \to \mathit{Tri}'\ B$, defined corecursively).

$$\mathit{redec}'\ f\ t := \langle f\ t, \mathit{frow}'\ t \rangle :: \mathit{redec}'\ f\ (\mathit{rest}'\ t)$$

We can finally show that this new version of the redecoration is equivalent to the previous one, modulo compatibility, using the conversion functions. We show that:

▶ **Lemma 25.**

$$\begin{aligned} &\forall f, \quad (\forall t\, t', t \cong t' \Rightarrow f\ t = f\ t') \\ &\Rightarrow \quad \forall t, \mathit{redec}'\ f\ t \cong \mathit{toStreamRep}\,(\mathit{redec}\,(f \circ \mathit{toStreamRep})\,(\mathit{fromStreamRep}\, t)) \end{aligned}$$

▶ **Lemma 26.**

$$\begin{aligned} &\forall f, \quad (\forall t\, t', t \simeq t' \Rightarrow f\ t = f\ t') \\ &\Rightarrow \quad \forall t, \mathit{redec}\ f\ t \simeq \mathit{fromStreamRep}\,(\mathit{redec}'\,(f \circ \mathit{fromStreamRep})\,(\mathit{toStreamRep}\, t)) \end{aligned}$$

▶ Remark. The compatibility hypotheses here are needed to work with Tri. Up to these extra requirements, the two conversion functions yield an isomorphism of comonads (the associated properties for top and top' are immediate by definition).

3.3 Simplifying redecoration again

As the representation of infinite triangles Tri' is only as a stream of streams, we can use standard functions on streams to define redecoration. Indeed, redecoration can be interpreted

as consisting of applying a function to each element of the diagonal of an infinite triangle, where each element of the diagonal is itself a triangle (iterated tails of the given triangle). We can thus decompose the redecoration operation into two steps: first transform the infinite triangle into a triangle of triangles and then apply the transformation function on the elements of the diagonal, as shown in Figure 11.

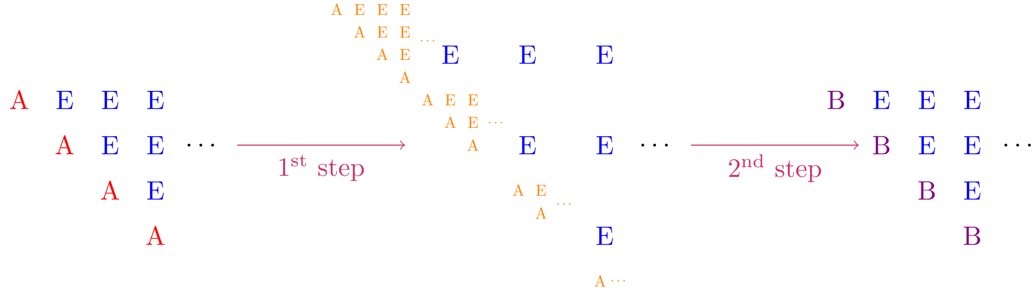


Figure 11 Idea of another definition of redecoration

▶ Remark. Figure 11 is only a visualization of what happens, and has to be taken lightly. In particular, all the elements of the diagonal of the middle triangle are infinite triangles, as we said. But, in order to visualize better what we do, their size seems to decrease since we cut out the first row of the previous element of the diagonal.

These two steps are then trivial to define on streams. Indeed, the first step consists of replacing all the elements of A by the corresponding iterated tail of the triangle itself. In fact, the information about the elements of E is redundant. Indeed, it is contained in the terms of the diagonal themselves (the row of elements of E "to the right" of an element of the diagonal is the first row of this element, minus the element of A). Therefore, we can omit them and only concentrate on the triangles. Thus, we need to obtain the stream of all the iterated tails of the initial triangle (see the first part of Figure 12). This is given by the classical *tails* operation defined below:

▶ **Definition 27** ($tails : \forall C.\, Str\, C \to Str(Str\, C)$, viewed coinductively)**.** $tails\, s := s :: tails(tl\, s)$

▶ Remark. The function *tails* has the signature of the comultiplication operation in a comonad based on *Str* according to the classical definition of comonads [8] (the term "comultiplication" is not used there, but only the letter δ that is dual to the multiplication of a monad). See Lemma 32 below for the constructive comonad based on *Str*.

In Figure 11, the second step only consists of applying f to all the elements of the diagonal. In fact, the first step corresponds to transforming t of type $Tri'\, A$ into

$$map\, \big(\lambda x.\langle x, frow'\, x\rangle\big)\, (tails\, t) \enspace ,$$

and the second one consists in transforming s of type $Tri'(Tri'\, A)$ into

$$map\, \big(\lambda\langle u, es\rangle.\, \langle f\, u, es\rangle\big)\, s \enspace .$$

We can alternatively see the transformation of t into $tails\, t$ as the first step, and the two successive *map* operations as the second step, which is therefore (by applying the functor law for *map* saying that *map*'s compose) performed by $map\, (lift'\, f)$, with $lift'$ defined as follows:

▶ **Definition 28** ($lift' : \forall A \forall B.\, (Tri'\, A \to B) \to Tri'\, A \to B \times Str\, E$)**.**

$$lift'\, f := \lambda x.\langle f\ x, frow'\ x\rangle$$

As for *rest* and *rest′*, *lift* and *lift′* are unrelated and belong to the respective point of view. This new version of the redecoration operation is shown in Figure 12.

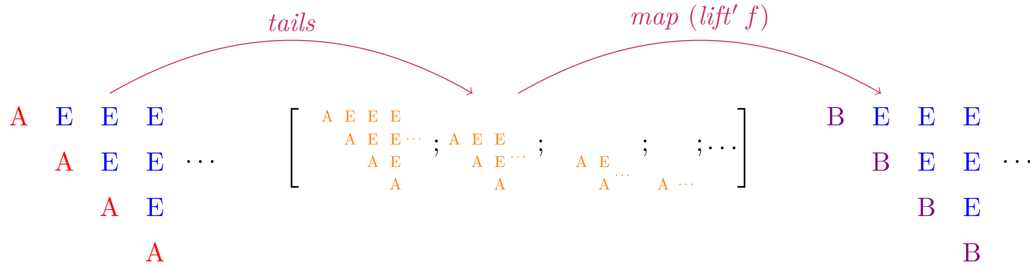


Figure 12 Another definition of redecoration

Thus, we define a new version of the redecoration operation as follows:

▶ **Definition 29** ($redec'_{alt} : \forall A \forall B. (Tri'\, A \to B) \to Tri'\, A \to Tri'\, B$).

$$redec'_{alt}\, f\, t := map\, (lift'\, f)\, (tails\, t)$$

One can then easily show that this operation is equivalent to the previous one:

▶ **Lemma 30.** $\forall A \forall B \forall (f : Tri'\, A \to B) \forall (t : Tri'\, A).\, redec'\, f\, t \equiv redec'_{alt}\, f\, t$

The proof is a straightforward coinduction.

It is interesting to note that here, we do not need the bisimulation relation defined on Tri'. We can directly use the standard relation on Str, $\equiv$. This should not be surprising. Indeed, here we only really manipulate streams. Those streams are made of pairs and we only manipulate the finite part of each pair (the first element). The second one is only a copy. Therefore the relation $\cong$ would be artificial here.

Let's continue abstracting and define $redec'_{gen}$ as follows:

▶ **Definition 31** ($redec'_{gen} : \forall A \forall B. (Str\, A \to B) \to Str\, A \to Str\, B$).

$$redec'_{gen}\, f\, s := map\, f\, (tails\, s)$$

As we remarked previously, *tails* has the signature of a comultiplication for a comonad (in the triple format [8]) based on *Str*, and it is well known that *map* is the functor (on morphisms) for *Str*. Therefore, $redec'_{gen}$ becomes the cobind operation of this comonad, generically. We do not develop this piece of constructive category theory here, but only state the result for this instance:

▶ **Lemma 32.** *The type transformation Str, the projection function hd and* $redec'_{gen}$ *form a constructive comonad with respect to* $\equiv$*.*

This section is inspired by Adriano [2] who suggested a redecoration function for Haskell lists just in this form. More precisely, a function `slide :: ([a] -> b) -> [a] -> [b]` was defined by `slide f = map f.tails`. Note that Haskell lists can be finite and infinite, thus this definition captured streams as well.

The function $redec'_{alt}$ is an instance of $redec'_{gen}$, i. e., the following lemma is trivial:

▶ **Lemma 33.** $\forall A \forall B \forall (f : Tri'\, A \to B)(t : Tri'\, A).\, redec'_{alt}\, f\, t = redec'_{gen}\, (lift'\, f)\, t$

Therefore, it is natural to show the three laws of comonads for $redec'_{alt}$ and the proofs are much simplified by the use of $redec'_{gen}$. In particular, we can show a kind of commutativity of $lift'$ with $redec'_{alt}$.

▶ **Lemma 34.** *The type transformation Tri', the projection function top' and $redec'_{alt}$ form a constructive comonad with respect to $\equiv$ (more precisely, the equivalence relation is $\equiv_{A\times Str\,E}$ for every A).*

▶ Remark. Through the functions *toStreamRep* and *fromStreamRep*, one can then transfer this comonad structure back to *Tri*. Since Lemma 25 and Lemma 26 require compatibility of f with bisimilarity, this will not even give a constructive weak comonad, but the first and third law have to be relativized to compatible f's as well. Still, this does not seem a problematic constraint. Anyway, Lemma 11 has been proved independently of streams.

4 Conclusion

In this paper we have presented various verifications of the redecoration algorithm for infinite triangles. We have first dualized directly the representation for finite triangles by a nested inductive datatype to obtain a nested coinductive datatype. In both cases, the triangles are visualized by their columns. We have implemented the corresponding redecoration algorithm *redec* (already available [1] in higher-order parametric polymorphism) and shown that we (only) obtained a constructive weak comonad (because of the compatibility hypothesis required). In this part, the redecoration algorithm, although deduced directly from the finite case, is quite tricky to manipulate because of the cutting and lifting it requires.

We then noticed that we could also consider the triangles by their rows, representing this time the triangles by purely coinductive datatypes, Tri', where we only took advantage of the existing type of streams (*Str*). This new visualization allowed us to define – keeping the same algorithmic idea as before – a function of redecoration $redec'$ already simpler than *redec* and equivalent to it, modulo compatibility. But taking advantage of this representation by streams, we can simplify again the redecoration algorithm, using only standard functions on streams. This new function $redec'_{alt}$ is fully equivalent to $redec'$. Generalizing again, we get nearly for free the cobind of the comonad *Str*, $redec'_{gen}$. This finally allows us to prove the three comonad laws for $redec'_{alt}$.

In short, we have shown that the redecoration function, which is a quite subtle operation if we translate it directly from the finite triangles, reduces to something very basic in the completely infinite (i. e., in both directions) view of the infinite triangles. In this case, it is much easier to work with only infinite elements than with partially finite ones in the sense of consisting of infinitely many finitely presented columns. In fact, the stream representation is even easier to manipulate than the representation of finite triangles, and the comonad laws even hold with less restrictions due to constructivity.

Notice that the row-based view would not have given new insights for finite triangles. Indeed, as they are symmetric, we would have obtained exactly the same representation as for the column-based approach, only perceived with interchanged roles of rows and columns.

As a final remark on the Coq side, the improved support for setoid rewriting and the class mechanism [12] has shown to be of great help for the formalization and verification decribed in this paper.

References

1. Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005.
2. Jorge Adriano. Answer to Markus Schnell's message `slide: useful function?` Haskell Mailing List, November 2002.
3. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.
4. Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer Verlag, 1998.
5. Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.
6. Nils Anders Danielsson. Beating the productivity checker using embedded languages. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *PAR*, volume 43 of *EPTCS*, pages 29–48, 2010.
7. Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In Claude Bolduc, Jules Desharnais, and Béchir Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer, 2010.
8. Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, second edition, 1998.
9. Ralph Matthes and Celia Picard. Formalization in Coq for this article, 2012. `www.irit.fr/~Celia.Picard/Coq/Redecoration/`.
10. Ralph Matthes and Martin Strecker. Verification of the redecoration algorithm for triangular matrices. In Furio Honsell, Marino Miculan, and Ivan Scagnetto, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Revised Selected Papers*, volume 4941 of *Lecture Notes in Computer Science*, pages 125–141. Springer Verlag, 2008.
11. Celia Picard. *Représentation coinductive des graphes.* PhD thesis, Université de Toulouse, 2012.
12. Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.
13. The Coq Development Team. The Coq Proof Assistant Reference Manual. INRIA.
14. Tarmo Uustalu and Varmo Vene. The dual of substitution is redecoration. In Kevin Hammond and Sharon Curtis, editors, *Scottish Functional Programming Workshop*, volume 3 of *Trends in Functional Programming*, pages 99–110. Intellect, 2001.