Guarded Cubical Type Theory: Path Equality for Guarded Recursion

Lars Birkedal[1], Aleš Bizjak[2], Ranald Clouston[3], Hans Bugge Grathwohl[4], Bas Spitters[5], and Andrea Vezzosi[6]

1 **Department of Computer Science, Aarhus University, Denmark**
2 **Department of Computer Science, Aarhus University, Denmark**
3 **Department of Computer Science, Aarhus University, Denmark**
4 **Department of Computer Science, Aarhus University, Denmark**
5 **Department of Computer Science, Aarhus University, Denmark**
6 **Department of Computer Science and Engineering, Chalmers University of Technology, Sweden**

Abstract

This paper improves the treatment of equality in guarded dependent type theory (GDTT), by combining it with cubical type theory (CTT). GDTT is an extensional type theory with guarded recursive types, which are useful for building models of program logics, and for programming and reasoning with coinductive types. We wish to implement GDTT with decidable type checking, while still supporting non-trivial equality proofs that reason about the extensions of guarded recursive constructions. CTT is a variation of Martin-Löf type theory in which the identity type is replaced by abstract paths between terms. CTT provides a computational interpretation of functional extensionality, is conjectured to have decidable type checking, and has an implemented type checker. Our new type theory, called guarded cubical type theory, provides a computational interpretation of extensionality for guarded recursive types. This further expands the foundations of CTT as a basis for formalisation in mathematics and computer science. We present examples to demonstrate the expressivity of our type theory, all of which have been checked using a prototype type-checker implementation, and present semantics in a presheaf category.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs, F.3.2 Semantics of Programming Languages

Keywords and phrases Guarded Recursion, Dependent Type Theory, Cubical Type Theory, Denotational Semantics, Homotopy Type Theory

Digital Object Identifier 10.4230/LIPIcs.CSL.2016.23

1 Introduction

Guarded recursion is a technique for defining and reasoning about infinite objects. Its applications include the definition of productive operations on data structures more commonly defined via coinduction, such as streams, and the construction of models of program logics for modern programming languages with features such as higher-order store and concurrency [6]. This is done via the type-former $\triangleright$, called 'later', which distinguishes data which is available immediately from data only available after some computation, such as the unfolding of a fixed-point. For example, guarded recursive streams are defined by the equation

$$\mathsf{Str}_A \;=\; A \times \triangleright \mathsf{Str}_A$$

rather than the more standard $\mathsf{Str}_A = A \times \mathsf{Str}_A$, to specify that the head is available now but the tail only later. The type for fixed-point combinators is then $(\triangleright A \to A) \to A$, rather

than the logically inconsistent $(A \to A) \to A$, disallowing unproductive definitions such as taking the fixed-point of the identity function.

Guarded recursive types were developed in a simply-typed setting by Clouston et al. [9], following earlier work [21, 3, 1], alongside a logic for reasoning about such programs. For large examples such as models of program logics, we would like to be able to formalise such reasoning. A major approach to formalisation is via *dependent types*, used for example in the proof assistants Coq [18] and Agda [22]. Bizjak et al. [8], following earlier work [5, 20], introduced guarded dependent type theory (GDTT), integrating the $\triangleright$ type-former into a dependently typed calculus, and supporting the definition of guarded recursive types as fixed-points of functions on universes, and guarded recursive operations on these types.

We wish to formalise non-trivial theorems about equality between guarded recursive constructions, but such arguments often cannot be accommodated within *intensional* Martin-Löf type theory. For example, we may need to be able to reason about the extensions of streams in order to prove the equality of different stream functions. Hence GDTT includes an equality reflection rule, which is well known to make type checking undecidable. This problem is close to well-known problems with functional extensionality [13, Sec. 3.1.3], and indeed this analogy can be developed. Just as functional extensionality involves mapping terms of type $(x : A) \to \mathsf{Id}\, B\, (f\, x)\, (g\, x)$ to proofs of $\mathsf{Id}\, (A \to B)\, f\, g$, extensionality for guarded recursion requires an extensionality principle for later types, namely the ability to map terms of type $\triangleright\, \mathsf{Id}\, A\, t\, u$ to proofs of $\mathsf{Id}\, (\triangleright A)\, (\mathsf{next}\, t)\, (\mathsf{next}\, u)$, where next is the constructor for $\triangleright$. These types are isomorphic in the intended model, the presheaf category $\widehat{\omega}$ known as the *topos of trees*, and so in GDTT their equality was asserted as an axiom. But in a calculus without equality reflection we cannot merely assert such axioms without losing canonicity.

Cubical type theory (CTT) [10] is a new type theory with a computational interpretation of functional extensionality but without equality reflection, and hence is a candidate for extension with guarded recursion, so that we may formalise our arguments without incurring the disadvantages of fully extensional identity types. CTT was developed primarily to provide a computational interpretation of the univalence axiom of Homotopy Type Theory [26]. The most important novelty of CTT is the replacement of inductively defined identity types by *paths*, which can be seen as maps from an abstract interval $\mathbb{I}$, and are introduced and eliminated much like functions. CTT can be extended with identity types which model all rules of standard Martin-Löf type theory [10, Sec. 9.1], but these are equivalent to path types, and in our paper it suffices to work with path types only. CTT has sound denotational semantics in (fibrations in) *cubical sets*, a presheaf category that is used to model homotopy types. Many basic syntactic properties of CTT, such as the decidability of type checking, and canonicity for base types, are yet to be proved, but a type checker has been implemented [1] that confers some confidence in such properties.

In Sec. 2 of this paper we propose *guarded cubical type theory* (GCTT), a combination of the two type theories[2] which supports non-trivial proofs about guarded recursive types via path equality, while retaining the potential for good syntactic properties such as decidable type-checking and canonicity. In particular, just as a term can be defined in CTT to witness functional extensionality, a term can be defined in GCTT to witness extensionality for later types. Further, we use elements of the interval of CTT to annotate fixed-points, and hence control their unfoldings. This ensures that fixed-points are path equal, but not judgementally equal, to their unfoldings, and hence prevents infinite unfoldings, an obvious source of

[1] `https://github.com/mortberg/cubicaltt`

[2] With the exception of the *clock quantification* of GDTT, which we leave to future work.

non-termination in any calculus with infinite constructions. The resulting calculus is shown via examples to be useful for reasoning about guarded recursive operations; we also view it as potentially significant from the point of view of CTT, extending its expressivity as a basis for formalisation.

In Sec. 3 we give sound semantics to this type theory via the presheaf category over the product of the categories used to define semantics for GDTT and CTT. This requires considerable work to ensure that the constructions of the two type theories remain sound in the new category, particularly the glueing and universe of CTT. The key technical challenge is to ensure that the $\triangleright$ type-former supports the *compositions* that all types must carry in the semantics of CTT.

We have implemented a prototype type-checker for this extended type theory [3], which provides confidence in the type theory's syntactic properties. All examples in this paper, and many others, have been formalised in this type checker.

For reasons of space many details and proofs are omitted from this paper, but are included in a technical appendix[4].

2 Guarded Cubical Type Theory

This section introduces guarded cubical type theory (GCTT), and presents examples of how it can be used to prove properties of guarded recursive constructions.

2.1 Cubical Type Theory

We first give a brief overview of *cubical type theory* [5] (CTT) [10]. We start with a standard dependent type theory with Π, Σ, natural numbers, and a Russell-style universe:

$$
\begin{array}{lcll}
\Gamma, \Delta & ::= & () \mid \Gamma, x : A & \text{Contexts} \\
t, u, A, B & ::= & x \mid \lambda x : A.t \mid t\, u \mid (x : A) \to B & \Pi\text{-types} \\
& \mid & (t, u) \mid t.1 \mid t.2 \mid (x : A) \times B & \Sigma\text{-types} \\
& \mid & 0 \mid \mathsf{s}\, t \mid \mathsf{natrec}\, t\, u \mid \mathsf{N} & \text{Natural numbers} \\
& \mid & \mathsf{U} & \text{Universe}
\end{array}
$$

We adhere to the usual conventions of considering terms and types up to α-equality, and writing $A \to B$, respectively $A \times B$, for non-dependent Π and Σ-types. We use the symbol '$=$' for judgemental equality.

The central novelty of CTT is its treatment of equality. Instead of the inductively defined identity types of intensional Martin-Löf type theory [17], CTT has *paths*. The paths between two terms t, u of type A form a sort of function space, intuitively that of continuous maps from some interval $\mathbb{I}$ to A, with endpoints t and u. Rather than defining the interval $\mathbb{I}$ concretely as the unit interval $[0, 1] \subseteq \mathbb{R}$, it is defined as the *free De Morgan algebra* on a discrete infinite set of names $\{i, j, k, \dots\}$. A De Morgan algebra is a bounded distributive lattice with an involution $1 - \cdot$ satisfying the De Morgan laws

$$1 - (i \wedge j) = (1 - i) \vee (1 - j), \qquad 1 - (i \vee j) = (1 - i) \wedge (1 - j).$$

The interval $[0, 1] \subseteq \mathbb{R}$, with min, max and $1 - \cdot$, is an example of a De Morgan algebra.

[3] http://github.com/hansbugge/cubicaltt/tree/gcubical
[4] http://cs.au.dk/~birke/papers/gdtt-cubical-technical-appendix.pdf
[5] http://www.cse.chalmers.se/~coquand/selfcontained.pdf is a self-contained presentation of CTT.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Path}\ A\ t\ u}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, i : \mathbb{I} \vdash t : A}{\Gamma \vdash \langle i \rangle\, t : \mathsf{Path}\ A\ t[0/i]\ t[1/i]} \qquad \frac{\Gamma \vdash t : \mathsf{Path}\ A\ u\ s \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash t\, r : A}$$

Figure 1 Typing rules for path types.

The syntax for elements of $\mathbb{I}$ is:

$$r, s \ ::= \ 0 \mid 1 \mid i \mid 1 - r \mid r \wedge s \mid r \vee s.$$

0 and 1 represent the endpoints of the interval. We extend the definition of contexts to allow introduction of a new name:

$$\Gamma, \Delta \ ::= \ \cdots \mid \Gamma, i : \mathbb{I}.$$

The judgement $\Gamma \vdash r : \mathbb{I}$ means that r draws its names from Γ. Despite this notation, $\mathbb{I}$ is not a first-class type. Path types and their elements are defined by the rules in Fig. 1. *Path abstraction*, $\langle i \rangle\, t$, and *path application*, $t\, r$, are analogous to λ-abstraction and function application, and support the familiar β-equality $(\langle i \rangle\, t)\, r = t[r/i]$ and η-equality $\langle i \rangle\, t\, i = t$. There are two additional judgemental equalities for paths, regarding their endpoints: given $p : \mathsf{Path}\ A\ t\ u$ we have $p\, 0 = t$ and $p\, 1 = u$.

Paths provide a notion of identity which is more extensional than that of intensional Martin-Löf identity types, as exemplified by the proof term for functional extensionality:

$$\mathsf{funext}\ f\ g \triangleq \lambda p.\, \langle i \rangle\, \lambda x.\, p\, x\, i \ : \ ((x : A) \to \mathsf{Path}\, B\ (f\, x)\ (g\, x)) \to \mathsf{Path}\ (A \to B)\ f\ g.$$

The rules above suffice to ensure that path equality is reflexive, symmetric, and a congruence, but we also need it to be transitive and, where the underlying type is the universe, to support a notion of transport. This is done via *(Kan) composition operations*.

To define these we need the *face lattice*, $\mathbb{F}$, defined as the free distributive lattice on the symbols $(i = 0)$ and $(i = 1)$ for all names i, quotiented by the relation $(i = 0) \wedge (i = 1) = 0_{\mathbb{F}}$. The syntax for elements of $\mathbb{F}$ is:

$$\varphi, \psi \ ::= \ 0_{\mathbb{F}} \mid 1_{\mathbb{F}} \mid (i = 0) \mid (i = 1) \mid \varphi \wedge \psi \mid \varphi \vee \psi.$$

As with the interval, $\mathbb{F}$ is not a first-class type, but the judgement $\Gamma \vdash \varphi : \mathbb{F}$ asserts that φ draws its names from Γ. We also have the judgement $\Gamma \vdash \varphi = \psi : \mathbb{F}$ which asserts the equality of φ and ψ in the face lattice. Contexts can be restricted by elements of $\mathbb{F}$:

$$\Gamma, \Delta \ ::= \ \cdots \mid \Gamma, \varphi.$$

Such a restriction affects equality judgements so that, for example, $\Gamma, \varphi \vdash \psi_1 = \psi_2 : \mathbb{F}$ is equivalent to $\Gamma \vdash \varphi \wedge \psi_1 = \varphi \wedge \psi_2 : \mathbb{F}$

We write $\Gamma \vdash t : A[\varphi \mapsto u]$ as an abbreviation for the two judgements $\Gamma \vdash t : A$ and $\Gamma, \varphi \vdash t = u : A$, noting the restriction with φ in the equality judgement. Now the composition operator is defined by the typing and equality rule

$$\frac{\Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A[0/i][\varphi \mapsto u[0/i]]}{\Gamma \vdash \mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0 : A[1/i][\varphi \mapsto u[1/i]]}.$$

A simple use of composition is to implement the transport operation for Path types

$$\mathsf{transp}^i\ A\ a\ \triangleq\ \mathsf{comp}^i\ A\ [0_{\mathbb{F}} \mapsto []]\ a\ :\ A[1/i],$$

where a has type $A[0/i]$. The notation [] stands for an empty *system*. In general a system is a list of pairs of faces and terms, and it defines an element of a type by giving the individual components at each face. We extend the syntax as follows:

$$t, u, A, B\ ::=\ \cdots \mid [\varphi_1\ t_1, \ldots, \varphi_n\ t_n].$$

Below we see two of the rules for systems; they ensure that the components of a system agree where the faces overlap, and that all the cases possible in the current context are covered:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash \varphi_1 \vee \ldots \vee \varphi_n = 1_{\mathbb{F}} : \mathbb{F} \qquad \Gamma, \varphi_i \vdash t_i : A \qquad \Gamma, \varphi_i \wedge \varphi_j \vdash t_i = t_j : A \qquad i, j = 1 \ldots n}{\Gamma \vdash [\varphi_1\ t_1, \ldots, \varphi_n\ t_n] : A}$$

$$\frac{\Gamma \vdash [\varphi_1\ t_1, \ldots, \varphi_n\ t_n] : A \qquad \Gamma \vdash \varphi_i = 1_{\mathbb{F}} : \mathbb{F}}{\Gamma \vdash [\varphi_1\ t_1, \ldots, \varphi_n\ t_n] = t_i : A}$$

We will shorten $[\varphi_1 \vee \ldots \vee \varphi_n \mapsto [\varphi_1\ t_1, \ldots, \varphi_n\ t_n]]$ to $[\varphi_1 \mapsto t_1, \ldots, \varphi_n \mapsto t_n]$.

A non-trivial example of the use of systems is the proof that Path is transitive; given $p : \mathsf{Path}\ A\ a\ b$ and $q : \mathsf{Path}\ A\ b\ c$ we can define

$$\mathsf{transitivity}\ p\, q \triangleq \langle i \rangle\, \mathsf{comp}^j\ A\ [(i = 0) \mapsto a, (i = 1) \mapsto q\, j]\ (p\, i)\ :\ \mathsf{Path}\ A\ a\ c.$$

This builds a path between the appropriate endpoints because we have the equalities $\mathsf{comp}^j\ A\ [1_{\mathbb{F}} \mapsto a]\ (p\, 0) = a$ and $\mathsf{comp}^j\ A\ [1_{\mathbb{F}} \mapsto q\, j]\ (p\, 1) = q\, 1 = c$.

For reasons of space we have omitted the descriptions of some features of CTT, such as glueing, and the further judgemental equalities for terms of the form $\mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0$ that depend on the structure of A.

2.2 Later Types

In Fig. 3 we present the 'later' types of guarded dependent type theory (GDTT) [8], with judgemental equalities in Figs. 4 and 5. Note that we do not add any new equation for the interaction of compositions with $\triangleright$; such an equation would be necessary if we were to add the eliminator prev for $\triangleright$, but this extension (which involves clock quantifiers) is left to further work. We delay the presentation of the fixed-point operation until the next section.

The typing rules use the *delayed substitutions* of GDTT, as defined in Fig. 2. Delayed substitutions resemble Haskell-style do-notation, or a delayed form of let-binding. If we have a term $t : \triangleright A$, we cannot access its contents 'now', but if we are defining a type or term that itself has some part that is available 'later', then this part *should* be able to use the contents of t. Therefore delayed substitutions allow terms of type $\triangleright A$ to be unwrapped by $\triangleright$ and next. As observed by Bizjak et al. [8] these constructions generalise the *applicative functor* [19] structure of 'later' types, by the definitions $\mathsf{pure}\ t \triangleq \mathsf{next}\, t$, and $f \circledast t \triangleq \mathsf{next}\, [f' \leftarrow f, t' \leftarrow t]\, .\, f'\, t'$, as well as a generalisation of the $\circledast$ operation from simple functions to Π-types. We here make the new observation that delayed substitutions can express the function $\widehat{\triangleright} : \triangleright \mathsf{U} \to \mathsf{U}$, introduced by Birkedal and Møgelberg [4] to express guarded recursive types as fixed-points on universes, as $\lambda u. \triangleright [u' \leftarrow u].u'$; see for example the definition of streams in Sec. 2.4.

$$\frac{\Gamma \vdash}{\vdash \cdot : \Gamma \nrightarrow \cdot} \qquad \frac{\vdash \xi : \Gamma \nrightarrow \Gamma' \qquad \Gamma \vdash t : \triangleright \xi.A}{\vdash \xi\,[x \leftarrow t] : \Gamma \nrightarrow \Gamma', x : A}$$

Figure 2 Formation rules for delayed substitutions.

$$\frac{\Gamma, \Gamma' \vdash A \qquad \vdash \xi : \Gamma \nrightarrow \Gamma'}{\Gamma \vdash \triangleright \xi.A} \qquad \frac{\Gamma, \Gamma' \vdash A : \mathsf{U} \qquad \vdash \xi : \Gamma \nrightarrow \Gamma'}{\Gamma \vdash \triangleright \xi.A : \mathsf{U}}$$

$$\frac{\Gamma, \Gamma' \vdash t : A \qquad \vdash \xi : \Gamma \nrightarrow \Gamma'}{\Gamma \vdash \mathsf{next}\, \xi.\, t : \triangleright \xi.A}$$

Figure 3 Typing rules for later types.

$$\frac{\vdash \xi\,[x \leftarrow t] : \Gamma \nrightarrow \Gamma', x : B \qquad \Gamma, \Gamma' \vdash A}{\Gamma \vdash \triangleright \xi\,[x \leftarrow t]\,.A = \triangleright \xi.A}$$

$$\frac{\vdash \xi\,[x \leftarrow t, y \leftarrow u]\,\xi' : \Gamma \nrightarrow \Gamma', x : B, y : C, \Gamma'' \qquad \Gamma, \Gamma' \vdash C \qquad \Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash A}{\Gamma \vdash \triangleright \xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A = \triangleright \xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A}$$

$$\frac{\vdash \xi : \Gamma \nrightarrow \Gamma' \qquad \Gamma, \Gamma', x : B \vdash A \qquad \Gamma, \Gamma' \vdash t : B}{\Gamma \vdash \triangleright \xi\,[x \leftarrow \mathsf{next}\, \xi.\, t]\,.A = \triangleright \xi.A[t/x]}$$

Figure 4 Type equality rules for later types (congruence and equivalence rules are omitted).

$$\frac{\vdash \xi\,[x \leftarrow t] : \Gamma \nrightarrow \Gamma', x : B \qquad \Gamma, \Gamma' \vdash u : A}{\Gamma \vdash \mathsf{next}\, \xi\,[x \leftarrow t]\,.\, u = \mathsf{next}\, \xi.\, u : \triangleright \xi.A}$$

$$\frac{\vdash \xi\,[x \leftarrow t, y \leftarrow u]\,\xi' : \Gamma \nrightarrow \Gamma', x : B, y : C, \Gamma'' \qquad \Gamma, \Gamma' \vdash C \qquad \Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash v : A}{\Gamma \vdash \mathsf{next}\, \xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.\, v = \mathsf{next}\, \xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.\, v : \triangleright \xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A}$$

$$\frac{\vdash \xi : \Gamma \nrightarrow \Gamma' \qquad \Gamma, \Gamma', x : B \vdash u : A \qquad \Gamma, \Gamma' \vdash t : B}{\Gamma \vdash \mathsf{next}\, \xi\,[x \leftarrow \mathsf{next}\, \xi.\, t]\,.\, u = \mathsf{next}\, \xi.\, u[t/x][\triangleright \xi.A[t/x]]} \qquad \frac{\Gamma \vdash t : \triangleright \xi.A}{\Gamma \vdash \mathsf{next}\, \xi\,[x \leftarrow t]\,.\, x = t[\triangleright \xi.A]}$$

Figure 5 Term equality rules for later types. We omit congruence and equivalence rules, and the rules for terms of type U, which reflect the type equality rules of Fig. 4.

$$\frac{\Gamma \vdash r : \mathbb{I} \qquad \Gamma, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{dfix}^r\, x.t : \triangleright A} \qquad \frac{\Gamma, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{dfix}^1\, x.t = \mathsf{next}\, t[\mathsf{dfix}^0\, x.t/x][\triangleright A]}.$$

Figure 6 Typing and equality rules for the delayed fixed-point

▶ **Example 1.** In GDTT it is essential that we can convert terms of type $\triangleright\xi.\, \mathsf{Id}_A\ t\ u$ into terms of type $\mathsf{Id}_{\triangleright\xi.A}\,(\mathsf{next}\,\xi.\,t)\,(\mathsf{next}\,\xi.\,u)$, as it is essential for *Löb induction*, the technique of proof by guarded recursion where we assume $\triangleright p$, deduce p, and hence may conclude p with no assumptions. This is achieved in GDTT by postulating as an axiom the following judgemental equality:

$$\mathsf{Id}_{\triangleright\xi.A}\,(\mathsf{next}\,\xi.\,t)\,(\mathsf{next}\,\xi.\,u) \;=\; \triangleright\xi.\,\mathsf{Id}_A\ t\ u \tag{1}$$

A term from left-to-right of (1) can be defined using the J-eliminator for identity types, but the more useful direction is right-to-left, as proofs of equality by Löb induction involve assuming that we later have a path, then converting this into a path on later types. In fact in GCTT we can define a term with the desired type:

$$\lambda p.\langle i\rangle\, \mathsf{next}\,\xi[p' \leftarrow p].\, p'\ i \;:\; (\triangleright\xi.\,\mathsf{Path}\ A\ t\ u) \to \mathsf{Path}\,(\triangleright\xi.A)\,(\mathsf{next}\,\xi.\,t)\,(\mathsf{next}\,\xi.\,u). \tag{2}$$

Note the similarity of this term and type with that of funext, for functional extensionality, presented on page 4. Indeed we claim that (2) provides a computational interpretation of extensionality for later types.

2.3 Fixed Points

In this section we complete the presentation of GCTT by addressing fixed points. In GDTT there are fixed-point constructions $\mathsf{fix}\,x.t$ with the judgemental equality $\mathsf{fix}\,x.t = t[\mathsf{next}\,\mathsf{fix}\,x.t/x]$. In GCTT we want decidable type checking, including decidable judgemental equality, and so we cannot admit such an unrestricted unfolding rule. Our solution it that fixed points should not be judgementally equal to their unfoldings, but merely *path equal*. We achieve this by decorating the fixed-point combinator with an interval element which specifies the position on this path. The 0-endpoint of the path is the stuck fixed-point term, while the 1-endpoint is the same term unfolded once. However this threatens canonicity for base types: if we allow stuck fixed-points in our calculus, we could have stuck closed terms $\mathsf{fix}^i\, x.t$ inhabiting N. To avoid this, we introduce the *delayed* fixed-point combinator dfix, which produces a term 'later' instead of a term 'now'. Its typing rule, and notion of equality, is given in Fig. 6. We will write $\mathsf{fix}^r\, x.t$ for $t[\mathsf{dfix}^r\, x.t/x]$, $\mathsf{fix}\, x.t$ for $\mathsf{fix}^0\, x.t$, and $\mathsf{dfix}\, x.t$ for $\mathsf{dfix}^0\, x.t$.

▶ **Lemma 2** (Canonical unfold lemma). *For any term $\Gamma, x : \triangleright A \vdash t : A$ there is a path between $\mathsf{fix}\, x.t$ and $t[\mathsf{next}\,\mathsf{fix}\, x.t/x]$, given by the term $\langle i\rangle\, \mathsf{fix}^i\, x.t$.*

Transitivity of paths (via compositions) ensures that $\mathsf{fix}\, x.t$ is path equal to any number of fixed-point unfoldings of itself.

A term a of type A is said to be a *guarded fixed point* of a function $f : \triangleright A \to A$ if there is a path from a to $f(\mathsf{next}\, a)$.

▶ **Proposition 3** (Unique guarded fixed points). *Any guarded fixed-point a of a term $f : \triangleright A \to A$ is path equal to $\mathsf{fix}\, x.f\, x$.*

Proof. Given $p : \mathsf{Path}\, A\ a\ (f\,(\mathsf{next}\, a))$, we proceed by Löb induction, i.e., by assuming $\mathsf{ih} : \triangleright(\mathsf{Path}\, A\ a\ (\mathsf{fix}\, x.f\, x))$. We can define a path

$$s \triangleq \langle i \rangle\ f(\mathsf{next}\,[q \leftarrow \mathsf{ih}]\,.\,q\,i)\ :\ \mathsf{Path}\, A\ (f(\mathsf{next}\, a))\ (f(\mathsf{next}\,\mathsf{fix}\, x.f\, x)),$$

which is well-typed because the type of the variable q ensures that $q\,0$ is judgementally equal to a, resp. $q\,1$ and $\mathsf{fix}\, x.f\, x$. Note that we here implicitly use the extensionality principle for later (2). We compose s with p, and then with the inverse of the canonical unfold lemma of Lem. 2, to obtain our path from a to $\mathsf{fix}\, x.f\, x$. We can write out our full proof term, where p^{-1} is the inverse path of p, as

$$\mathsf{fix}\,\mathsf{ih}\,.\,\langle i \rangle\,\mathsf{comp}^j\ A\ [(i=0) \mapsto p^{-1}, (i=1) \mapsto f(\mathsf{dfix}^{1-j}\, x.f\, x)]\ (f(\mathsf{next}\,[q \leftarrow \mathsf{ih}]\,.\,q\,i)).$$ ◀

2.4 Programming and Proving with Guarded Recursive Types

In this section we show some simple examples of programming with guarded recursion, and prove properties of our programs using Löb induction.

Streams. The type of guarded recursive streams in GCTT, as with GDTT, are defined as fixed points on the universe:

$$\mathsf{Str}_A\ \triangleq\ \mathsf{fix}\, x.A \times \triangleright[y \leftarrow x].y$$

Note the use of a delayed substitution to transform a term of type $\triangleright\,\mathsf{U}$ to one of type U, as discussed at the start of Sec. 2.2. Desugaring to restate this in terms of dfix, we have

$$\mathsf{Str}_A\ =\ A \times \triangleright[y \leftarrow \mathsf{dfix}^0\, x.A \times \triangleright[y \leftarrow x].y].y$$

The head function $\mathsf{hd} : \mathsf{Str}_A \to A$ is the first projection. The tail function, however, cannot be the second projection, since this yields a term of type

$$\triangleright\left[y \leftarrow \mathsf{dfix}^0\, x.A \times \triangleright\,[y \leftarrow x]\,.y\right]\,.y \tag{3}$$

rather than the desired $\triangleright\,\mathsf{Str}_A$. However we are not far off; $\triangleright\,\mathsf{Str}_A$ is judgementally equal to $\triangleright\left[y \leftarrow \mathsf{dfix}^1\, x.A \times \triangleright\,[y \leftarrow x]\,.y\right]\,.y$, which is the same term as (3), apart from endpoint 1 replacing 0. The canonical unfold lemma (Lem. 2) tells us that we can build a path in U from Str_A to $A \times \triangleright\,\mathsf{Str}_A$; call this path $\langle i \rangle\,\mathsf{Str}_A^i$. Then we can transport between these types:

$$\mathsf{unfold}\ s \triangleq \mathsf{transp}^i\ \mathsf{Str}_A^i\ s \qquad\qquad \mathsf{fold}\ s \triangleq \mathsf{transp}^i\ \mathsf{Str}_A^{1-i}\ s$$

Note that the compositions of these two operations are path equal to identity functions, but not judgementally equal. We can now obtain the desired tail function $\mathsf{tl} : \mathsf{Str}_A \to \triangleright\,\mathsf{Str}_A$ by composing the second projection with unfold, so $\mathsf{tl}\ s \triangleq (\mathsf{unfold}\ s).2$. Similarly we can define the stream constructor cons (written infix as ::) by using fold:

$$\mathsf{cons} \triangleq \lambda a, s.\,\mathsf{fold}\,(a, s)\ :\ A \to \triangleright\,\mathsf{Str}_A \to \mathsf{Str}_A\,.$$

We now turn to higher order functions on streams. We define $\mathsf{zipWith} : (A \to B \to C) \to \mathsf{Str}_A \to \mathsf{Str}_B \to \mathsf{Str}_C$, the stream function which maps a binary function on two input streams to produce an output stream, as

$$\mathsf{zipWith}\ f \triangleq \mathsf{fix}\, z.\lambda s_1, s_2.f\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\ ::\ \mathsf{next}\left[\begin{array}{l} z' \leftarrow z \\ t_1 \leftarrow \mathsf{tl}\ s_1 \\ t_2 \leftarrow \mathsf{tl}\ s_2 \end{array}\right].\, z'\, t_1\, t_2.$$

Of course zipWith is definable even with simple types and $\triangleright$, but in GCTT we can go further and prove properties about the function:

▶ **Proposition 4** (zipWith preserves commutativity). *If $f : A \to A \to B$ is commutative, then* $\mathsf{zipWith}\ f : \mathsf{Str}_A \to \mathsf{Str}_A \to \mathsf{Str}_B$ *is commutative.*

Proof. Let $\mathsf{c} : (a_1 : A) \to (a_2 : A) \to \mathsf{Path}\, B\ (f\, a_1\, a_2)\ (f\, a_2\, a_1)$ witness commutativity of f. We proceed by Löb induction, i.e., by assuming

$$\mathsf{ih} : \triangleright\, ((s_1 : \mathsf{Str}_A) \to (s_2 : \mathsf{Str}_A) \to \mathsf{Path}\, B\ (\mathsf{zipWith}\ f\, s_1\, s_2)\ (\mathsf{zipWith}\ f\, s_2\, s_1))\,.$$

Let $i : \mathbb{I}$ be a fresh name, and $s_1, s_2 : \mathsf{Str}_A$. Our aim is to construct a stream v which is $\mathsf{zipWith}\ f\, s_1\, s_2$ when substituting 0 for i, and $\mathsf{zipWith}\ f\, s_2\, s_1$ when substituting 1 for i. An initial attempt at this proof is the term

$$v \triangleq \mathsf{c}\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\,i\ ::\ \mathsf{next}\left[\begin{array}{l} q \leftarrow \mathsf{ih} \\ t_1 \leftarrow \mathsf{tl}\ s_1 \\ t_2 \leftarrow \mathsf{tl}\ s_2 \end{array}\right].\, q\, t_1\, t_2\, i\ :\ \mathsf{Str}_B,$$

which is equal to

$$f\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\ ::\ \mathsf{next}\left[\begin{array}{l} t_1 \leftarrow \mathsf{tl}\ s_1 \\ t_2 \leftarrow \mathsf{tl}\ s_2 \end{array}\right].\ \mathsf{zipWith}\ f\, t_1\, t_2$$

when substituting 0 for i, which is $\mathsf{zipWith}\ f\, s_1\, s_2$, but *unfolded once*. Similarly, $v[1/i]$ is $\mathsf{zipWith}\ f\, s_2\, s_1$ unfolded once. Let $\langle j \rangle\, \mathsf{zipWith}^j$ be the canonical unfold lemma associated with zipWith (see Lem. 2). We can now finish the proof by composing v with (the inverse of) the canonical unfold lemma. Diagrammatically, with i along the horizontal axis and j along the vertical:

$$\begin{array}{ccc} \mathsf{zipWith}\ f\, s_1\, s_2 & \dashrightarrow & \mathsf{zipWith}\ f\, s_2\, s_1 \\ \uparrow{\scriptstyle \mathsf{zipWith}^{1-j}\ f\, s_1\, s_2} & & \uparrow{\scriptstyle \mathsf{zipWith}^{1-j}\ f\, s_2\, s_1} \\ f\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\ :: \\ \mathsf{next}\left[\begin{array}{l} t_1 \leftarrow \mathsf{tl}\ s_1 \\ t_2 \leftarrow \mathsf{tl}\ s_2 \end{array}\right].\ \mathsf{zipWith}\ f\, t_1\, t_2 & \xrightarrow[v]{} & f\,(\mathsf{hd}\ s_2)\,(\mathsf{hd}\ s_1)\ :: \\ & & \mathsf{next}\left[\begin{array}{l} t_2 \leftarrow \mathsf{tl}\ s_2 \\ t_1 \leftarrow \mathsf{tl}\ s_1 \end{array}\right].\ \mathsf{zipWith}\ f\, t_2\, t_1 \end{array}$$

The complete proof term, in the language of the type checker, can be found in Appendix A. ◀

Guarded recursive types with negative variance. A key feature of guarded recursive types are that they support *negative* occurrences of recursion variables. This is important for applications to models of program logics [6]. Here we consider a simple example of a negative variance recursive type, namely $\mathsf{Rec}_A \triangleq \mathsf{fix}\, x.(\triangleright[x' \leftarrow x].x') \to A$, which is path equal to $\triangleright\, \mathsf{Rec}_A \to A$. As a simple demonstration of the expressiveness we gain from negative guarded recursive types, we define a guarded variant of Curry's Y combinator:

$$\begin{array}{lcll} \Delta & \triangleq & \lambda x.f(\mathsf{next}[x' \leftarrow x].((\mathsf{unfold}\, x')x)) & : \quad \triangleright\, \mathsf{Rec}_A \to A \\ \mathsf{Y} & \triangleq & \lambda f.\Delta(\mathsf{next}\,\mathsf{fold}\,\Delta) & : \quad (\triangleright A \to A) \to A, \end{array}$$

where fold and unfold are the transports along the path between Rec_A and $\triangleright\, \mathsf{Rec}_A \to A$. As with zipWith, Y can be defined with simple types and $\triangleright$ [1]; what is new to GCTT is that we can also prove properties about it:

▶ **Proposition 5** (Y is a guarded fixed-point combinator). *$\mathsf{Y}\, f$ is path equal to $f\,(\mathsf{next}(\mathsf{Y}\, f))$, for any $f : \triangleright A \to A$. Therefore, by Prop. 3,* Y *is path equal to* fix.

Proof. $\mathsf{Y}\, f$ simplifies to $f\,(\mathsf{next}(\mathsf{unfold}\,(\mathsf{fold}\,\Delta)\,(\mathsf{next}\,\mathsf{fold}\,\Delta)))$, and $\mathsf{unfold}\,(\mathsf{fold}\,\Delta)$ is path equal to Δ. A congruence over this path yields our path between $\mathsf{Y}\, f$ and $f(\mathsf{next}(\mathsf{Y}\, f))$. ◀

3 Semantics

In this section we sketch the semantics of GCTT. The semantics is based on the category $\widehat{\mathcal{C} \times \omega}$ of presheaves on the category $\mathcal{C} \times \omega$, where $\mathcal{C}$ is the *category of cubes* [10] and ω is the poset of natural numbers. The category of cubes is the opposite of the Kleisli category of the free De Morgan algebra monad on finite sets. More concretely, given a countably infinite set of names $i, j, k, \ldots$, $\mathcal{C}$ has as objects finite sets of names I, J. A morphism $I \to J \in \mathcal{C}$ is a *function* $J \to \mathbf{DM}\,(I)$, where $\mathbf{DM}\,(I)$ is the free De Morgan algebra with generators I.

Following the approach of Cohen et al. [10], contexts of GCTT will be interpreted as objects of $\widehat{\mathcal{C} \times \omega}$. Types in context Γ will be interpreted as pairs (A, c_A) of a presheaf A on the category of elements of Γ and a *composition structure* c_A. We call such a pair a *fibrant* type.

To aid in defining what a composition structure is, and in showing that composition structure is preserved by all the necessary type constructions, we will make use of the internal language of $\widehat{\mathcal{C} \times \omega}$ in the form of *dependent predicate logic*; see for example Phoa [24, App. I].

A type of GCTT in context Γ will then be interpreted as a pair of a type $\Gamma \vdash A$ in the internal language of $\widehat{\mathcal{C} \times \omega}$, and a composition structure c_A, where c_A is a term in the internal language of a specific type $\Phi(\Gamma; A)$, which we define below after introducing the necessary constructs. Terms of GCTT will be interpreted as terms of the internal language. We use *categories with families* [12] as our notion of a model. Due to space limits we omit the precise definition of the category with families here, and refer to the online technical appendix.

The semantics is split into several parts, which provide semantics at different levels of generality.

1. We first show that every presheaf topos with a non-trivial internal De Morgan algebra $\mathbb{I}$ satisfying the disjunction property can be used to give semantics to the subset of the cubical type theory CTT without glueing and the universe. We further show that, for any category $\mathbb{D}$, the category of presheaves on $\mathcal{C} \times \mathbb{D}$ has an interval $\mathbb{I}$, which is the inclusion of the interval in presheaves over the category of cubes $\mathcal{C}$.
2. We then extend the semantics to include glueing and universes. We show that the topos of presheaves $\mathcal{C} \times \mathbb{D}$ for any category $\mathbb{D}$ with an initial object can be used to give semantics to the entire cubical type theory.
3. Finally, we show that the category of presheaves on $\mathcal{C} \times \omega$ gives semantics to delayed substitutions and fixed points. Using these and some additional properties of the delayed substitutions we show in the internal language of $\widehat{\mathcal{C} \times \omega}$ that $\triangleright \xi.A$ has composition whenever A has composition.

Combining all three, we give semantics to GCTT in $\widehat{\mathcal{C} \times \omega}$.

3.1 Model of CTT Without Glueing and the Universe

Let $\mathcal{E}$ be a topos with a natural numbers object, and let $\mathbb{I}$ be a De Morgan algebra internal to $\mathcal{E}$ which satisfies the *finitary disjunction property*, i.e.,

$$(i \vee j) = 1 \implies (i = 1) \vee (j = 1), \quad \text{and} \quad \neg(0 = 1).$$

Faces. Using the interval $\mathbb{I}$ we define the type $\mathbb{F}$ as the image of the function $\cdot = 1 : \mathbb{I} \to \Omega$, where Ω is the subobject classifier. More precisely, $\mathbb{F}$ is the subset type

$$\mathbb{F} \triangleq \{p : \Omega \mid \exists (i : \mathbb{I}), p = (i = 1)\}$$

We will implicitly use the inclusion $\mathbb{F} \to \Omega$. The following lemma states in particular that the inclusion is compatible with all the lattice operations, so omitting it is justified. The disjunction property is crucial for validity of this lemma.

▶ **Lemma 6.**

- $\mathbb{F}$ *is a lattice for operations inherited from* Ω.
- *The corestriction* $\cdot = 1 : \mathbb{I} \to \mathbb{F}$ *is a lattice homomorphism. It is not injective in general.*

Given $\Gamma \vdash \varphi : \mathbb{F}$, we write $[\varphi] \triangleq \mathrm{Id}_{\mathbb{F}}(\varphi, \top)$. Given $\Gamma \vdash A$ and $\Gamma \vdash \varphi : \mathbb{F}$ a *partial element* of type A of *extent* φ is a term t of type $\Gamma \vdash t : \Pi(p : [\varphi]).A$. If we are in a context with $p : [\varphi]$, then we will treat such a partial element t as a term of type A, leaving implicit the application to the proof p, i.e., we will treat t as $t\,p$. We will often write $\Gamma, [\varphi]$ instead of $\Gamma, p : [\varphi]$ when we do not mention the proof term p explicitly in the rest of the judgement. This is justified since inhabitants of $[\varphi]$ are unique up to judgemental equality (recall that dependent predicate logic is a logic over an extensional dependent type theory). Given $\Gamma, p : [\varphi] \vdash B$ we write B^{φ} for the dependent function space $\Pi(p : [\varphi]).B$ and again leave the proof p implicit.

For a term $\Gamma, p : [\varphi] \vdash u : A$ we define $A[\varphi \mapsto u] \triangleq \Sigma(a : A).\,(\mathrm{Id}_A(a, u))^{\varphi}$.

Compositions. Faces allow us to define the type of *compositions* $\Phi(\Gamma; A)$. Homotopically, compositions allow us to put a lid on a box [10]. Given $\Gamma \vdash A$ we define the corresponding type of compositions as

$$\Phi(\Gamma; A) \triangleq \Pi(\gamma : \mathbb{I} \to \Gamma)(\varphi : \mathbb{F})\,(u : \Pi(i : \mathbb{I}).\,(A(\gamma(i)))^{\varphi})\,. \\ A(\gamma(0))[\varphi \mapsto u(0)] \to A(\gamma(1))[\varphi \mapsto u(1)].$$

Here we treat the context Γ as a closed type. This is justified because there is a canonical bijection between contexts and closed types of the internal language. The notation $A(\gamma(i))$ means substitution along the (uncurried) γ.

Due to lack of space we do not show how the standard constructs of the type theory are interpreted. We only sketch how the following composition term is interpreted in terms of the composition in the model.

$$\frac{\Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A[0/i][\varphi \mapsto u[0/i]]}{\Gamma \vdash \mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0 : A[1/i][\varphi \mapsto u[1/i]]}.$$

By assumption we have c_A of type $\Phi(\Gamma, i : \mathbb{I}; A)$ and u and a_0 are interpreted as terms in the internal language of the corresponding types. The interpretation of composition is the term

$$\gamma : \Gamma \vdash c_A\,(\lambda(i : \mathbb{I}).(\gamma, i))\,\varphi\,(\lambda(i : \mathbb{I})(p : [\varphi]).u)\,a_0 : A(\gamma(1))[\varphi \mapsto u(1)]$$

where we have omitted writing the proof $u(0) = a_0$ on $[\varphi]$.

Concrete models. The category of cubical sets has an internal interval type satisfying the disjunction property [10]. It is the functor mapping $I \in \mathcal{C}$ to $\mathbf{DM}\,(I)$. Since the theory of a De Morgan algebra with $0 \neq 1$ and the disjunction property is geometric [16, Section X.3] we have that for any topos $\mathcal{F}$ and geometric morphism $\varphi : \mathcal{F} \to \widehat{\mathcal{C}}$, $\varphi^*(\mathbb{I}) \in \mathcal{F}$ is a De Morgan algebra with the disjunction property[6]. In particular, given any category $\mathbb{D}$ there is a projection functor $\pi : \mathcal{C} \times \mathbb{D} \to \mathcal{C}$ which induces the (essential) geometric morphism $\pi^* \dashv \pi_* : \widehat{\mathcal{C} \times \mathbb{D}} \to \widehat{\mathcal{C}}$, where π^* is precomposition with π, and π_* takes limits along $\mathbb{D}$.

[6] A statement very close to this can be used as a characterisation of $\widehat{\mathcal{C}}$: this topos classifies the geometric theory of flat De Morgan algebras [25].

Summary. With the semantic structures developed thus far we can give semantics to the subset of CTT without glueing and the universe.

3.2 Adding Glueing and the Universe

The glueing construction [10, Sec. 6] is used to prove both fibrancy and, subsequently, univalence of the universe of fibrant types. Concretely, given

$$\Gamma \vdash \varphi : \mathbb{I} \qquad \Gamma, [\varphi] \vdash T \qquad \Gamma \vdash A \qquad \Gamma \vdash w : (T \to A)^{\varphi}$$

we define the type Glue $[\varphi \mapsto (T, w)]\ A$ in two steps. First we define the type[7]

$$Glue'_{\Gamma}(\varphi, T, A, w) \triangleq \sum_{a:A} \sum_{t:T^{\varphi}} \prod_{p:[\varphi]} wp(tp) = a.$$

For this type we have the following property $\Gamma, [\varphi] \vdash T \cong Glue'_{\Gamma}(\varphi, T, A, w)$. However, we need an equality, not an isomorphism, to obtain the correct typing rules. The technical appendix provides a general strictification lemma which allows us to define the type *Glue*.

To show that the type Glue $[\varphi \mapsto (T, w)]\ A$ is fibrant we need to additionally assume that the map $\varphi \mapsto \lambda\_.\varphi : \mathbb{F} \to (\mathbb{I} \to \mathbb{F})$ has an internal right adjoint $\forall$. Such a right adjoint exists in all toposes $\widehat{\mathcal{C} \times \mathbb{D}}$, for any small category $\mathbb{D}$ with an initial object.

Universe of fibrant types. Given a (Grothendieck) universe $\mathfrak{U}$ in the meta-theory, the Hofmann-Streicher universe [14] $\mathcal{U}^{\omega}$ in $\widehat{\mathcal{C} \times \omega}$ maps (I, n) to the set of functors valued in $\mathfrak{U}$ on the category of elements of $y(I, n)$, where y is the Yoneda embedding. As in Cohen et al. [10] we define the universe of *fibrant* types $\mathcal{U}^{\omega}_f$ by setting $\mathcal{U}^{\omega}_f(I, n)$ to be the set of fibrant types in context $y(I, n)$. The universe $\mathcal{U}^{\omega}_f$ satisfies the rules

$$\frac{\Gamma \vdash a : \mathcal{U} \qquad \vdash \mathbf{c} : \Phi(\Gamma; \mathrm{El}(a))}{\Gamma \vdash (\!|a, \mathbf{c}|\!) : \mathcal{U}_f} \qquad \frac{\Gamma \vdash a : \mathcal{U}_f}{\Gamma \vdash \mathrm{El}(a)} \qquad \frac{\Gamma \vdash a : \mathcal{U}_f}{\vdash \mathrm{Comp}(a) : \Phi(\Gamma; \mathrm{El}(a))}$$

Using the glueing operation, one shows that the universe of fibrant types is *itself* fibrant and, moreover, that it is univalent.

3.3 Adding the Later Type-Former

We now fix the site to be $\mathcal{C} \times \omega$. From the previous sections we know that $\widehat{\mathcal{C} \times \omega}$ gives semantics to CTT. The new constructs of GDTT are the $\triangleright$ type-former and its delayed substitutions, and guarded fixed points. Continuing to work in the internal language, we first show that the internal language of $\widehat{\mathcal{C} \times \omega}$ can be extended with these constructions, allowing interpretation of the subset of the type theory GDTT without clock quantification [8]. Due to lack of space we omit the details of this part, but do remark that $\triangleright$ is defined as

$$(\triangleright(X))(I, n) \begin{cases} \{\star\} & \text{if } n = 0 \\ X(I, m) & \text{if } n = m + 1 \end{cases}$$

The essence of this definition is that $\triangleright$ depends only on the "ω component" and ignores the "$\mathcal{C}$ component". Verification that all the rules of GDTT are satisfied is therefore very similar to the verification that the topos $\widehat{\omega}$ is a model of the same subset of GDTT.

[7] This type is already present in Kapulkin at al. [15, Thm 3.4.1].

The only additional property we need now is that $\triangleright$ preserves compositions, in the sense that if we have a delayed substitution $\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$ and a type $\Gamma, \Gamma' \vdash A$ together with a closed term $\mathbf{c}_A$ of type $\Phi(\Gamma, \Gamma'; A)$ then we can construct $\mathbf{c}'_{\triangleright\xi.A}$ of type $\Phi(\Gamma; \triangleright\xi.A)$.

The following lemma uses the notion of a type $\Gamma \vdash A$ being *constant with respect to* ω. This notion is a natural generalisation to types-in-context of the property that a presheaf is in the image of the functor π^*. We refer to the online technical appendix for the precise definition. Here we only remark that the interval type $\mathbb{I}$ is constant with respect to ω, as is the type $\Gamma \vdash [\varphi]$ for any term $\Gamma \vdash \varphi : \mathbb{F}$.

▶ **Lemma 7.** *Assume* $\Gamma \vdash A$, $\Gamma, \Gamma', x : A \vdash B$ *and* $\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$, *and further that* A *is constant with respect to* ω. *Then the following two types are isomorphic*

$$\Gamma \vdash \triangleright\xi.\Pi(x : A).B \cong \Pi(x : A).\triangleright\xi.B \tag{4}$$

and the canonical morphism $\lambda f.\lambda x.\, \mathsf{next}\,[\xi, f' \leftarrow f]\,.\, f'\,x$ *from left to right is an isomorphism.*

▶ **Corollary 8.** *If* $\Gamma \vdash \varphi : \mathbb{F}$ *then we have an isomorphism of types*

$$\Gamma \vdash \triangleright\xi.\Pi(p : [\varphi]).B \cong \Pi(x : [\varphi]).\triangleright\xi.B. \tag{5}$$

▶ **Lemma 9** ($\triangleright\xi$-types preserve compositions)**.** *If* $\triangleright\xi.A$ *is a well-formed type in context* Γ *and we have a composition term* $\mathbf{c}_A : \Phi(\Gamma, \Gamma'; A)$, *then there is a composition term* $\mathbf{c} : \Phi(\Gamma; \triangleright\xi.A)$.

Proof. We show the special case with an empty delayed substitution. For the more general proof we refer to the technical appendix. Assume we have a composition $\mathbf{c}_A : \Phi(\Gamma; A)$. Our goal is to find a term $\mathbf{c} : \Phi(\Gamma; \triangleright A)$, so we first introduce some variables:

$$\gamma : \mathbb{I} \to \Gamma \qquad \varphi : \mathbb{F} \qquad u : \Pi(i : \mathbb{I}).\,((\triangleright A)(\gamma\, i))^{\varphi} \qquad a_0 : (\triangleright A)(\gamma\, 0)[\varphi \mapsto u\, 0].$$

Using the isomorphisms from Cor. 8 and Lem. 7 we obtain a term $\tilde{u} : \triangleright(\Pi(i : \mathbb{I}).(A(\gamma\, i))^{\varphi})$ isomorphic to u. We can now – almost – write the term

$$\mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\, \mathbf{c}_A\, \gamma\, \varphi\, u'\, a'_0 \;:\; \triangleright(A(\gamma\, 1)), \tag{$*$}$$

what is missing is to check that $a'_0 = u'\, 0$ on the extent φ, so that we can legally apply $\mathbf{c}_A$; this is equivalent to saying that the type $\triangleright\,[u' \leftarrow \tilde{u}, a'_0 \leftarrow a_0]\,.\,\mathrm{Id}_{A(\gamma\, 0)}(a'_0, u'\, 0)^{\varphi}$ is inhabited. We transform this type as follows:

$$\begin{aligned} \triangleright\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\mathrm{Id}(a'_0, u'\, 0)^{\varphi} &\cong \left(\triangleright\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\mathrm{Id}(a'_0, u'\, 0)\right)^{\varphi} && \text{(Cor. 8)} \\ &= \left(\mathrm{Id}(\mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].a'_0, \mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].u'\, 0)\right)^{\varphi} \\ &= (\mathrm{Id}(a_0, u\, 0))^{\varphi}, \end{aligned}$$

where the last equality uses that $\tilde{u}$ is defined using the inverse of $\lambda f \lambda x.\, \mathsf{next}\,[f' \leftarrow f]\,.\, f'\,x$ (Lem. 7). By assumption it is the case that $(\mathrm{Id}(a_0, u\, 0))^{\varphi}$ is inhabited, and therefore $(*)$ is well-defined. It remains only to check that $(*)$ is equal to $u\, 1$ on the extent φ, but this follows from the equalities of $\mathbf{c}_A$ and by the definition of $\tilde{u}$ (Lem. 7). Assuming φ, we have

$$\mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\, \mathbf{c}_A\, \gamma\, \varphi\, u'\, a'_0 = \mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\, u'\, 1 = u\, 1. \qquad \blacktriangleleft$$

Summary. In this section we have highlighted the key ingredients that go into a sound interpretation of GCTT in $\widehat{\mathcal{C} \times \omega}$. For the precise statement of the interpretation of all the constructs, and the soundness theorem, we refer to the online technical appendix.

4 Conclusion

In this paper we have made the following contributions:

- We introduce guarded cubical type theory (GCTT), which combines features of cubical type theory (CTT) and guarded dependent type theory (GDTT). The path equality of CTT is shown to support reasoning about extensional properties of guarded recursive operations, and we use the interval of CTT to constrain the unfolding of fixed-points.
- We show that CTT can be modelled in any presheaf topos with an internal non-trivial De Morgan algebra with the disjunction property, an operator $\forall$, and a universe of fibrant types. Most of these constructions are done via the internal logic. We then show that a class of presheaf models of the form $\widehat{\mathcal{C} \times \mathbb{D}}$, for any category $\mathbb{D}$ with an initial object, satisfy the above axioms and hence gives rise to a model of CTT.
- We give semantics to GCTT in the topos of presheaves over $\mathcal{C} \times \omega$.

Further work. We wish to establish key *syntactic properties* of GCTT, namely decidable type-checking and canonicity for base types. Our prototype implementation establishes some confidence in these properties.

We wish to further extend GCTT with *clock quantification* [3], such as is present in GDTT. Clock quantification allows for the controlled elimination of the later type-former, and hence the encoding of first-class coinductive types via guarded recursive types. The generality of our approach to semantics in this paper should allow us to build a model by combining cubical sets with the presheaf model of GDTT with multiple clocks [7]. The main challenges lie in ensuring decidable type checking (GDTT relies on certain rules involving clock quantifiers which seem difficult to implement), and solving the *coherence problem* for clock substitution.

Finally, some higher inductive types, like the truncation, can be added to CTT. We would like to understand how these interact with $\triangleright$.

Related work. Another type theory with a computational interpretation of functional extensionality, but without equality reflection, is observational type theory (OTT) [2]. We found CTT's prototype implementation, its presheaf semantics, and its interval as a tool for controlling unfoldings, most convenient for developing our combination with GDTT, but extending OTT similarly would provide an interesting comparison.

Spitters [25] used the interval of the internal logic of cubical sets to model identity types. Coquand [11] defined the composition operation internally to obtain a model of type theory. We have extended both these ideas to a full model of CTT. Recent independent work by Orton and Pitts [23] axiomatises a model for CTT without a universe, again building on Coquand [11]. With the exception of the absence of the universe, their development is more general than ours. Our semantic developments are sufficiently general to support the sound addition of guarded recursive types to CTT.

Acknowledgements. We gratefully acknowledge our discussions with Thierry Coquand, and the comments of our reviewers. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU). Aleš Bizjak was supported in part by a Microsoft Research PhD grant.

References

1 Andreas Abel and Andrea Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *APLAS*, pages 140–158, 2014.

2 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV*, pages 57–68, 2007.

3 Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208, 2013.

4 Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *LICS*, pages 213–222, 2013.

5 Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *LMCS*, 8(4), 2012. doi:10.2168/LMCS-8(4:1)2012.

6 Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, pages 119–132, 2011.

7 Aleš Bizjak and Rasmus Ejlers Møgelberg. A model of guarded recursion with clock synchronisation. In *MFPS*, pages 83–101, 2015.

8 Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *FoSSaCS*, pages 20–35, 2016.

9 Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In *FoSSaCS*, pages 407–421, 2015.

10 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. Unpublished, 2016.

11 Thierry Coquand. Internal version of the uniform Kan filling condition. Unpublished, 2015. URL: http://www.cse.chalmers.se/~coquand/shape.pdf.

12 Peter Dybjer. Internal type theory. In *TYPES'95*, pages 120–134, 1996.

13 Martin Hofmann. *Extensional constructs in intensional type theory.* Springer, 1997.

14 Martin Hofmann and Thomas Streicher. Lifting Grothendieck universes. Unpublished, 1999. URL: http://www.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf.

15 Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. *arXiv:1211.2851*, 2012.

16 Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic.* Springer, 1992. doi:10.1007/978-1-4612-0927-0.

17 Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium 1973*, pages 73–118, 1975.

18 The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0. URL: http://coq.inria.fr.

19 Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Programming*, 18(1):1–13, 2008.

20 Rasmus Ejlers Møgelberg. A type theory for productive coprogramming via guarded recursion. In *CSL-LICS*, 2014.

21 Hiroshi Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

22 Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

23 Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. In *CSL*, 2016.

24 Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, LFCS, University of Edinburgh, 1992.

25 Bas Spitters. Cubical sets as a classifying topos. *TYPES*, 2015.
26 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Institute for Advanced Study, 2013. URL: `http://homotopytypetheory.org/book`.

A zipWith Preserves Commutativity

We provide a formalisation of Sec. 2.4 which can be verified by our type checker[8].

```
module zipWith_preserves_comm where

Id (A : U) (a0 a1 : A) : U = IdP (⟨i⟩ A) a0 a1
data nat = Z | S (n : nat)

-- Streams of natural numbers
StrF (S : ▷ U) : U = (n : nat) * ▷ [S' ← S] S'

Str : U = fix (StrF Str)

-- The canonical unfold lemma for Str
StrUnfoldPath : Id U Str (StrF (next Str))
 = ⟨i⟩ StrF (dfix U StrF [(i=1)])

unfoldStr (s : Str) : (n : nat) * ▷ Str
 = transport StrUnfoldPath s

foldStr (s : (n : nat) * ▷ Str) : Str
 = transport (⟨i⟩ StrUnfoldPath @ -i) s

cons (n : nat) (s : ▷ Str) : Str = foldStr (n, s)
head (s : Str) : nat = s.1
tail (s : Str) : ▷ Str = (unfoldStr s).2

-- Defining zipWith
zipWithF (f : nat → nat → nat) (rec : ▷ (Str → Str → Str))
 : Str → Str → Str
 = (λ (s1 s2 : Str) →
      (cons (f (head s1) (head s2))
            (next [zipWith' ← rec, s1' ← tail s1 , s2' ← tail s2]
                  zipWith' s1' s2')))

zipWith (f : nat → nat → nat) : Str → Str → Str
 = fix (zipWithF f zipWith)

zipWithUnfoldPath (f : nat → nat → nat)
 : Id (Str → Str → Str)
      (zipWith f)
      (zipWithF f (next (zipWith f)))
 = ⟨i⟩ zipWithF f (dfix (Str → Str → Str) (zipWithF f) [(i=1)])

-- Commutativity property
comm (f : nat → nat → nat) : U = (m n : nat) → Id nat (f m n) (f n m)

-- zipWith preserves commutativity.
zipWith_preserves_comm (f : nat → nat → nat) (c : comm f)
 : (s1 s2 : Str) → Id Str (zipWith f s1 s2) (zipWith f s2 s1)
 = fix
   (λ (s1 s2 : Str) →
      ⟨i⟩ comp (⟨_⟩ Str)
               (cons (c (head s1) (head s2) @ i)
                     (next [q ← zipWith_preserves_comm
                           ,t1 ← tail s1
                           ,t2 ← tail s2]
                           q t1 t2 @ i))
               [(i=0) → ⟨j⟩ zipWithUnfoldPath f @ -j s1 s2
               ,(i=1) → ⟨j⟩ zipWithUnfoldPath f @ -j s2 s1])
```

[8] This file, among other examples, is available in the `gctt-examples` folder in the type-checker repository.