Guarded Recursive Type Theory

Hans Bugge Grathwohl

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Guarded Recursive Type Theory

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Hans Bugge Grathwohl
Friday 30th September, 2016

Abstract

Currently, total languages such as Agda and Coq employ syntactic checks to enforce productivity of programs that manipulate potentially infinite data. With guarded recursive types we can instead rely on the type system for ensuring productivity, thus allowing for a more modular programming style.

Guarded recursion is both a technique for ensuring productivity, and an abstraction of the step-indexing technique used for modelling programming languages with sophisticated features, such as recursive types, higher-order store, and concurrency. This dissertation is about enriching type theory with guarded recursion in the form of guarded recursive types. These types can be utilised for programming and reasoning with coinductive types through Atkey and McBride style clock quantifiers.

In each chapter we present a new type theory with guarded recursive types, reflecting different challenges arising when describing guarded recursive types.

In the first chapter we present a simple type theory with guarded recursive and coinductive types, along with a program logic which can be used to reason about the operational behaviour of the terms.

In the next chapter we present an extensional dependent type theory with guarded recursive and coinductive types, in which we can encode both proofs and programs.

The third chapter focuses on the treatment of propositional equality. Here we present a type theory that combines guarded recursive types with cubical type theory, a new type theory which provides a computational interpretation of Voevodsky's univalence axiom, and thus also of function extensionality.

The last chapter presents preliminary work on an alternative presentation of dependent type theory with guarded recursive and coinductive types, for which we can define a reduction relation on terms and types which we conjecture is strongly normalising.

Resumé

I dag benytter totale programmeringssprog, såsom Agda og Coq, sig af syntaktiske tjek for at sikre produktivitet af programmer der manipulerer potentielt uendeligt data. Med beskyttede rekursive typer kan vi i stedet sikre produktivitet igennem typesystemet og dermed åbne op for en mere modulær programmeringsstil.

Beskyttet rekursion er både en teknik til at garantere produktivitet, og en abstraktion af step-indexing-teknikken som bruges til at modellere programmeringssprog med sofistikerede features, så som rekursive typer, højereordenslager og concurrency. Denne afhandling handler om at berige typeteori med beskyttet rekursion i form af beskyttede rekursive typer. Disse typer kan benyttes til at programmere med, og ræsonnere om, koinduktive typer via urkvantorer à la Atkey og McBride.

I hvert kapitel præsenteres en ny typeteori med beskyttede rekursive typer, og reflekterer dermed forskellige udfordringer der opstår når disse typer beskrives.

I det første kapitel præsenteres en simpel typeteori med beskyttede rekursive og koinduktive typer, sammen med en programlogik der kan benyttes til at ræsonnere om den operationelle opførsel af termer.

I det næste kapitel præsenteres en ekstensionel afhængig typeteori med beskyttede rekursive og koinduktive typer, hvori vi kan repræsentere både beviser og programmer.

Det tredje kapitel fokuserer på behandlingen af propositionel lighed. Her præsenteres en typeteori der kombinerer beskyttede rekursive typer med kubisk typeteori, en ny typeteori som giver en beregnelighedsfortolkning af Voevodskys univalensaksiom, og derfor også af funktionel ekstensionalitet.

I det sidste kapitel præsenteres foreløbigt arbejde omhandlende en alternativ præsentation af afhængig type teori med beskyttede rekursive og koinduktive typer, hvortil vi kan definere en reduktionsrelation på termer og typer, hvilken vi formoder er stærkt normaliserende.

Acknowledgments

My three years in Aarhus have been great, and for that I have many people to thank. First and foremost I am grateful to my supervisor Lars Birkedal for accepting me as his student (despite my lack of computer science background) and for his expert guidance over the years.

By working for Olivier Danvy I have learned much about teaching – his attitude towards education, and devotion to all of his students, has been very inspiring.

I am thankful to all of my coauthors for valuable collaboration. Thanks to Andrea Vezzosi for hosting me at Chalmers, Gothenburg, and for the many hours spend experimenting with implementations. Thanks to Patrick Bahr and Rasmus Møgelberg for all the times they hosted me at the IT University, Copenhagen.

I am grateful to Aleš Bizjak, Ranald Clouston, and to my brother Bjørn for reading and providing valuable feedback on parts of this document.

Thanks to everyone in the Logic and Semantics group for making it a great work environment, and to all of my friends in Aarhus for all the good times.

And last, but not least: Thank you Roos, for deciding to move your life to Denmark with me, for putting up with me, and for letting me put up with you.

Hans Bugge Grathwohl,
Aarhus, Friday 30th September, 2016.

Contents

Chapter 1

Introduction

Type systems can be used to ensure that programs have certain nice properties. With *guarded recursive types* one can ensure that programs manipulating potentially infinite data – such as coinductive data types – are productive, i.e., they will always yield well-defined outputs when provided with well-defined inputs.

Certain type systems that are sufficiently strong are called *type theories*, and can be used to formalise logical and mathematical proofs and propositions via the Curry-Howard correspondence. In this setting guarded recursive types capture a reasoning principle called *Löb induction* which can be used to reason about potentially infinite data, and has been used as an abstraction of step-indexing to reason about models of programming languages. Productivity is essential to ensure logical consistency of the type theory.

This PhD dissertation is about adding guarded recursive types to type theory.

The reader is assumed to be familiar with type theory, and some familiarity with coinductive types and categorical models will be an advantage.

1.1 Motivation

Guarded recursion is a technique for solving recursive type equations. It does not matter whether the type variable occurs positively or negatively, as long as it appears *guarded* by a modality $\triangleright$, which we will call 'later'. It can be seen as an abstract account of the step-indexing technique introduced by Appel and McAllester [6]. In recent years solutions of guarded recursive domain equations have been used to model programming languages with sophisticated features such as recursive types, higher-order store, and concurrency [7, 13, 36]. Such models are used for software verification, a process where some form of automation is desirable due to the inevitably long and complicated proof obligations. For this reason there are a number of implementations of program logics in the proof assistant Coq [63], for example:

The ModuRes framework [80] for reasoning about concurrent higher-order imperative programs using solutions to recursive domain equations in the category of complete bounded bisected ultrametric spaces, and more recently an implementation of the Iris program logic [16] which uses the same category. Common to these implementations is that the guarded recursion happens inside models embedded in the proof assistant. Birkedal et al. [14] and Birkedal and Møgelberg [10] show that guarded recursion can be defined in terms of the *topos of trees*, and that this model can be used to justify the existence of a type theory with guarded recursive types. The hope is that a well-developed type theory with guarded recursive types can be used as foundation for a proof assistant, making it simpler to implement methods for reasoning about programs based on guarded domain equations.

Paviotti et al. [72] and Møgelberg and Paviotti [68] have constructed models of the programming languages PCF and FPC entirely in the guarded recursive type theory presented in Chapter 3 of this dissertation. This supports our conjecture that guarded recursive types can be a useful tool for formalising models of programming languages.

As observed by Nakano [69] and Atkey and McBride [8], guarded recursive types is also an attractive approach to ensuring productivity of coinductive definitions. This will be the focus of the following overview of guarded recursive types, as it is generally simpler to talk about, whereas formalisations in type theory of models of programming languages typically require a larger apparatus, e.g., inductive families.

1.2 An Overview of Guarded Recursive Types

In this section we will give an informal overview of guarded recursive types as they are treated in this dissertation.

1.2.1 Recursion, Corecursion, and Guarded Recursion

Guarded recursive types are to guarded recursion what recursive types are to recursion. Recursion is, informally, the concept of *self-reference*. We use self-reference in programming languages when we let functions call themselves. This recursion can be used to traverse structured data. In some settings "recursion" is restricted to least fixed-point constructions involving inductive data types, such as natural numbers, lists, and trees, as these forms of data are themselves least fixed-point constructions. This opens up for a dual concept: "corecursion" which is when functions are defined as greatest fixed-points, involving coinductive data types. We will use recursion to mean any kind of self-reference.

Guarded recursion[1] is a specific kind of recursion, where the self-reference

[1] Not to be confused with *guarded recursive datatype constructors* [85].

is "protected" (or *guarded*) by a modality which ensures that the recursion is "sensible". The most prevalent metaphor, and indeed the one used throughout this dissertation, is a temporal one: When we look at an object *now* then it might be defined in terms of the object itself, but the self-reference can only be accessed *at a later time*. Definitions by guarded recursion are unique fixed points.

1.2.2 Productivity and Syntactic Checks

The canonical example of infinite structures in computer science is *streams*. A stream can be thought of as a never-ending list of data. Lists are inductive data types – which for example can be expressed by saying that lists of natural numbers is the initial algebra for the Sets functor $1+\mathbb{N}\times(-)$ – whereas streams are typically seen as *coinductive data types*: the final coalgebra for the functor $\mathbb{N}\times(-)$.

In pseudocode of a lazily evaluated language we can give a recursive definition of streams of natural numbers:

```
Str = Nat × Str
```

Then we can define some basic stream functions:

```
(::) : Nat → Str → Str
n :: s = ⟨n, s⟩

head : Str → Nat
head = π1

tail : Str → Str
tail = π2
```

where $\langle -,- \rangle$, π_1, and π_2 are the constructor and destructors of the product type $A \times B$. We can use recursion to define streams. As a simple example, consider the stream consisting of only 1's:

```
ones : Str
ones = 1 :: ones
```

But when we write definitions in this system it is easy to write something nonsensical, like

```
foo : Str
foo = tail foo
```

This definition is bad because the equation `foo = tail foo` does not have a unique solution in `Str`, as opposed to `ones = 1 :: ones`. We also say `ones` is *productive*, because any finite prefix of it can be computed in finite time by unfolding the definition, whereas the definition of `foo` is unproductive.

So how do we ensure that a definition is productive? One solution, utilised by total programming languages such as Agda [70] and Coq [63], is to perform a syntactic check of the definition ensuring that the *recursive call occurs directly underneath a constructor* in the syntax tree.[2] If we in this case consider `(::)` to be a constructor, then this check will indeed allow `ones` but disallow `foo`. It is indeed the case that such a check will only allow productive definitions [41].

Now consider the `map` function for natural number streams:

```
map : (Nat → Nat) → Str → Str
map f s = f (head s) :: map f (tail s)
```

Since the recursive call (`map f`) occurs directly below the constructor, this definition will be allowed by a syntactic check. But if we use this function in the definition of another stream like so:

```
nats : Str
nats = 0 :: map succ nats
```

where `succ : Nat → Nat` is the successor function, then the syntactic productivity check would disallow it, because the recursive call `nats` is applied to the function `map succ` instead of the constructor. By unfolding the definition it is evident that there is a unique solution to the equation, namely the stream $0, 1, 2, 3, \ldots$ of natural numbers. But the productivity of `nats` depends on the fact that `map succ` is well-behaved. By replacing this function call we can indeed get an unproductive definition:

```
bad : Str
bad = 0 :: tail bad
```

Any stream starting with 0 will be a solution to this equation, so `bad` is unproductive and therefore rightly disallowed by the syntactic check. Note that the types of `tail` and `map succ` are identical, so the type system has no way of distinguishing these definition. But from these examples it is clear that there is an important difference between them. Guarded recursive types provide a means of expressing this difference.

1.2.3 Guarded Recursive Types

With guarded recursive types productivity is ensured by the type system. The core of guarded recursive types is the modality that guards the type of a recursive call. It originates from Nakano's calculus [69], but we follow Appel et al. [7] by writing $\triangleright$ and using the name 'later'. We will think of $\triangleright A$ as the type of terms which *will be* of type A after one time step. One can always

[2] The algorithms used by these languages employ some further heuristics to allow more definitions.

save a term for later, therefore we will have a constructor for $\triangleright$ called next:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \triangleright A}$$

When writing a definition using *guarded recursion* – say we want to define a term of type A – then the type of the recursive call will be guarded by a 'later', i.e., $\triangleright A$. We express this by having a *guarded fixed-point combinator* in the language

$$\frac{\Gamma, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.t : A}$$

for which we have that

$$\mathsf{fix}\, x.t = t[\mathsf{next}\, \mathsf{fix}\, x.t/x].$$

If we revisit the stream programming examples from the previous section, this time in pseudocode of a language with guarded recursive types, then the type of (guarded) streams of natural numbers will be

```
Str = Nat × ▷Str
```

and the basic stream operations will be identical save for their types

```
(::) : Nat → ▷Str → Str
n :: s = ⟨n, s⟩

head : Str → Nat
head = π1

tail : Str → ▷Str
tail = π2
```

Note that the `tail` of a stream is only going to be a stream at the next tick of the clock. The definition of `ones` will be identical to before:

```
ones : Str
ones = 1 :: ones▷
```

where `ones`$^{\triangleright}$ is the guarded recursive call (which will be encoded by the guarded fixed-point combinator). It is well-typed because the guarded recursive call has the type `▷Str` as required by the stream constructor `(::)`. But the unproductive term `foo`

```
foo : Str
foo = foo▷
```

is *not* well-typed, because the type of the right-hand side of the equality sign does not match the supposed type of the left-hand side (`▷Str` vs. `Str`).

In order to define the map function on guarded streams we need some more structure on $\triangleright$. If we have a term which will be a function later, $t : \triangleright(A \to B)$, and another term which will be a suitable input to this function later, $u : \triangleright A$, then we want a way to 'schedule' a function application. This is done with the 'delayed application' operator, $\circledast$:

$$\frac{\Gamma \vdash t : \triangleright(A \to B) \qquad \Gamma \vdash u : \triangleright A}{\Gamma \vdash t \circledast u : \triangleright B}$$

As this choice of notation suggests, $\triangleright$ is an *applicative functor* [64], and in particular the following equality will be satisfied:

$$\mathsf{next}\, t \circledast \mathsf{next}\, u = \mathsf{next}(t\, u).$$

Now we can define `map`:

```
map : (Nat → Nat) → Str → Str
map f s = f (head s) :: (map f)▸ ⊛ (tail s)
```

The guarded recursive call in this definition is `map f`, which has type `▸(Str → Str)`, and since `tail s` must have type `▸Str`, then `map f ⊛ (tail s)` has type `▸Str`, and thus the definition is well-typed. Likewise, we can define the stream of natural numbers

```
nats : Str
nats = 0 :: next (map succ) ⊛ nats▸
```

while there is no way to make the unproductive definition

```
bad : Str
bad = 0 :: tail bad▸
```

well-typed, because the type of `tail` means that there will always be a 'later' too much in the type.

1.2.4 Denotational Semantics

We will here briefly discuss the intuition of a model of guarded recursive types – a full definition of a model of a simple type theory with guarded recursion can be found in Section 2.3. Earlier Nakano's guarded recursive type theory had been modelled using complete bounded ultrametric spaces [11]. However, we will be using the *topos of trees* to model our guarded recursive types, as first done by Birkedal et al. [14]. As the name suggests, this is a *topos* which means that it can also be used to model dependent type theory. The topos of trees is defined as $\mathrm{PSh}(\omega)$, the presheaf category over ω, where ω is the ordinal containing the natural numbers starting from 1. Thus, an object A of this category is a family of sets A_n along with restriction maps $r_n : A_{n+1} \to A_n$:

$$A_1 \xleftarrow{r_1} A_2 \xleftarrow{r_2} A_3 \xleftarrow{r_3} A_4 \longleftarrow \cdots$$

and the morphisms $f : A \to B$ are natural transformations between such diagrams

$$\begin{array}{ccccccccc} A_1 & \longleftarrow & A_2 & \longleftarrow & A_3 & \longleftarrow & A_4 & \longleftarrow & \cdots \\ \downarrow{\scriptstyle f_1} & & \downarrow{\scriptstyle f_2} & & \downarrow{\scriptstyle f_3} & & \downarrow{\scriptstyle f_4} & & \\ B_1 & \longleftarrow & B_2 & \longleftarrow & B_3 & \longleftarrow & B_4 & \longleftarrow & \cdots \end{array}$$

where all the squares commute. Our types will be modelled with objects of $\mathrm{PSh}(\omega)$, which we will think of as a sequence of *approximations* of the type. The type of guarded streams of natural numbers will for example be modelled with the object consisting of increasingly long tuples of natural numbers:

$$\mathbb{N} \xleftarrow{\pi_1} \mathbb{N} \times \mathbb{N} \xleftarrow{\pi_1} (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \longleftarrow \cdots$$

We can think of this like so: To begin with we can only observe the first element of the stream, then after one time-step we can see the second element, and so on.

Like any topos, $\mathrm{PSh}(\omega)$ has an internal higher-order logic which we can use to reason about its objects. In Section 2.4 we describe how to use this internal logic to reason about programs with guarded recursive types.

1.2.5 Causality and Coinductive Types

A simply typed language with $\triangleright$, next, $\circledast$, fix, and a way of defining guarded recursive types allows us to express a lot of programs with potentially infinite data. But not all productive definitions fit into this framework. Consider the stream function `every2nd`, which returns a stream containing every second element of its input stream:

```
every2nd : Str → Str
every2nd s = head s :: every2nd (tail (tail s))
```

There is no way that we can sprinkle this definition with nexts and $\circledast$'s in order to make it a well-typed guarded recursive definition. This is because the two uses of `tail` in the definition create two $\triangleright$'s, which we have no means of turning into the single $\triangleright$ that is needed. Note however that this definition *does* pass the syntactic productivity check that we described earlier, and thus it would readily be accepted as a valid coinductive stream definition in languages such as Coq and Agda.

The `every2nd` function is an example of an *acausal* stream function. A stream function is *causal* if the n first elements of the output stream relies at most on the n first elements of the input stream, and otherwise it is *acausal*. With the guarded recursive types we have described so far, all stream functions we define must necessarily be causal. In certain contexts this is desirable, for example if the streams are supposed to model real-world streams of data

such as keyboard input, or stock prices. A version of these guarded recursive types has been used by Krishnaswami and Benton [56] to model *reactive programming*.

In order to program with acausal stream functions we extend the system with a way of looking at *all of a type at once*. Recall that we interpret the type of guarded streams as the following diagram:

$$\mathbb{N} \xleftarrow{\pi_1} \mathbb{N} \times \mathbb{N} \xleftarrow{\pi_1} (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \longleftarrow \cdots$$

This diagram has a limit, $\mathbb{N}^\omega$, in the category of sets consisting of the actual streams, which we can embed into the topos of trees like so:

$$\mathbb{N}^\omega \xleftarrow{\mathrm{id}} \mathbb{N}^\omega \xleftarrow{\mathrm{id}} \mathbb{N}^\omega \longleftarrow \cdots$$

In Chapter 2 we add a modality to the type system, written ■ and called 'everything now' (or 'constant'), which gives us the limit of the approximation of a type. So while `Str = Nat × ▹Str` is the type of *guarded* streams (or approximations of streams), `■Str` is the type of *coinductive* streams, on which we can also define acausal functions. There is a list of acausal example programs using the ■ modality starting on page 31 of this dissertation. The typing rules for ■ can be found on page 27 of this dissertation.

In Chapter 3 we use an alternative to the ■ modality, namely Atkey and McBride [8] style *clock quantifiers*, which also provides a way of programming (and reasoning with) coinductive types.

Another technique for ensuring productivity through types is *sized types* [2, 4, 45], where recursion is reduced to a well-founded induction on a 'size index' annotated on the types. A discussion of the relation between guarded recursive types and sized types can be found in Section 2.6.1.

1.2.6 Dependent Types and Delayed Substitutions

Getting guarded recursive types to work well in combination with a simple type theory is only a step on the way to incorporating guarded recursive types into a fully fledged dependent type theory. The first significant difference between a simple type theory and a dependent type theory is the function type. In a dependent type theory, the simple function type $A \to B$ is replaced with the dependent function type $(x : A) \to B$ (also called a Π-type), where x may occur free in B. This means that the typing rule for function application now includes a substitution on the result type:

$$\frac{\Gamma \vdash t : (x : A) \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[u/x]}$$

This immediately raises an issue with our 'later' types: what does $t \circledast u$ mean now? If t has the type $\triangleright((x : A) \to B)$ and u has type $\triangleright A$, then we should be

able to do a delayed function application. But it is unclear what to do with the free x in B, since we do not yet have anything to substitute for it. This is our motivation for introducing *delayed substitutions*. A delayed substitution is annotated on $\triangleright$ and next, and is of the form $\triangleright[x \leftarrow u].A$ and $\mathsf{next}[x \leftarrow u].t$. Then we will have the equalities

$$\triangleright[x \leftarrow \mathsf{next}\, u].A = \triangleright A[u/x],$$
$$\mathsf{next}[x \leftarrow \mathsf{next}\, u].t = \mathsf{next}\, t[u/x].$$

It can be thought of as a form of let-binding:[3] $\triangleright(\mathsf{let}\,\mathsf{next}\, x = u\,\mathsf{in}\, A)$.

With delayed substitution we can provide a typing rule for $\circledast$ that works with the dependent product type:

$$\frac{\Gamma \vdash t : \triangleright((x : A) \to B) \qquad \Gamma \vdash u : \triangleright A}{\Gamma \vdash t \circledast u : \triangleright[x \leftarrow u].B}$$

However, it turns out that with delayed substitutions we do not even need $\circledast$ as a primitive construct in the language, as $t \circledast u$ is equivalent to

$$\mathsf{next}[x \leftarrow t, y \leftarrow u].x\, y.$$

Delayed substitutions are treated in Chapter 3.

1.2.7 Identity Types and Löb Induction

Looking at the guarded fixed-point combinator through the Curry-Howard correspondence it corresponds to a reasoning principle known as *Löb induction*: from $\triangleright P \Rightarrow P$ we can conclude P. This becomes especially useful when our language includes *identity types*, $t =_A u$, and the following extensionality principle for $\triangleright$:

$$\triangleright(t =_A u) \;\cong\; (\mathsf{next}\, t =_{\triangleright A} \mathsf{next}\, u). \tag{1.1}$$

In Section 3.3 and 3.5 there are examples where we use guarded recursion on identity types to prove properties of programs, but here we will confine ourselves to a small toy example: Let `generate` be the function which from a base point and a generator function returns a guarded stream:

```
generate : Nat → (Nat → Nat) → Str
generate n f = n :: generate▸ ⊛ next (f n) ⊛ next f
```

Now use `generate` to define a representation of the stream of ones, while defining an alternative directly using guarded recursion:

```
ones = generate 1 (λn.n)
ones' = 1 :: ones'▸
```

[3] For Haskellers it might be helpful to think of them as a kind of *applicative do* notation.

Note, that by unfolding the (implicit) guarded fixed-point combinator once we get that `ones` is definitionally equal to `1 :: next ones`, and `ones'` is definitionally equal to `1 :: next ones'`. We will use Löb induction to show that `ones` and `ones'` are propositionally equal. For this we will use a term `eta` for constructing equalities of streams (the definition of which is not important) and a witness of (1.1), `ext`:

```
eta : (head s₁ = head s₂) → (tail s₁ = tail s₂) → (s₁ = s₂)
ext : ▹(t=u) → (next t = next u)
```

Now, since `tail ones` unfolds to `next ones`, and `tail ones'` unfolds to `next ones'`, we can write out our proof, defined by guarded recursion:

```
ones-are-equal : ones = ones'
ones-are-equal = eta refl (ext ones-are-equal▸)
```

This proof helps illustrate an important point: Guarded recursive types do not work well in combination with intensional identity types à la Martin-Löf. The intensional identity type is typically viewed as an *inductive family* [38], generated by the constructors $\mathsf{refl}_t : t = t$, and therefore a closed term like `ones-are-equal` which *doesn't* reduce to refl is problematic. We have violated the *canonicity property* of type theory, because we can use the induction principle for identity types (also known as J) to produce a stuck closed term of any inhabited type. Here we show how to make a closed term of type `Nat` which does not reduce to a natural number: Assume that we have the following witness of the J rule.

```
J : (P : (x y : A) → (x = y) → Type) →
    ((x : A) → P x x reflₓ) → (x y : A) → (p : x = y)
    → P x y p
```

Then we can write the following program

```
NaN : Nat
NaN = J (λ _ _ _. Nat) (λ _. 60) ones ones' ones-are-equal
```

which will always have `J` as its outermost connective since it requires `ones-are-equal` to be `refl` before it will reduce.

The *guarded dependent type theory* presented in Chapter 3 is extensional, i.e., identity types are not distinguished from definitional equalities. This means that we avoid the issues arising from using an intensional equality, but one consequence of this choice is that type-checking is undecidable.

1.2.8 Bisimulation and Guarded Recursion

The traditional proof method for coinductive types is proof by *bisimulation*. A bisimulation is a binary relation $R \subseteq X \times Y$ between coalgebras X and Y fulfilling certain requirements – an introduction to coalgebras and bisimulation can be found in Jacobs and Rutten [48]. The arguably most important

property of bisimulations is the *coinductive proof principle*: If $R \subseteq Z \times Z$ is a bisimulation on a final coalgebra Z, then $R(x, y)$ implies $x = y$.

A bisimulation on streams is a relation $\sim$ that fulfils

$$s_1 \sim s_2 \iff \begin{array}{l} \mathsf{head}\, s_1 = \mathsf{head}\, s_2, \text{ and} \\ \mathsf{tail}\, s_2 \sim \mathsf{tail}\, s_2. \end{array}$$

In Chlipala [24] there is a tutorial describing how to define $\sim$ in Coq using coinduction, and how to show bisimilarity of streams. However, showing that the coinductive proof principle holds for $\sim$ requires *external* reasoning, i.e., reasoning in the model of Coq. This means that we cannot use $s_1 \sim s_2$ to obtain a proof of $s_1 = s_2$. In practice one could decide to use the bisimulation relation as the 'identity type for streams'. But this means that one does not get the benefits of identity types, e.g., transports and congruence properties.

With guarded recursive types we can bridge this gap between the external and internal reasoning. In guarded recursive type theory a coinductive type is defined as the limit of its approximations, and when we work with a coinductive type we always 'unbox' it and work with the approximations instead. Therefore, to define a bisimulation on streams we will first define a *guarded* bisimulation relation $\sim^g$ on *guarded* streams:

$$s_1 \sim^g s_2 = (\mathsf{head}\, s_1 = \mathsf{head}\, s_2) \times \triangleright \left[\begin{array}{l} \sim' \leftarrow (\sim^g)^{\triangleright} \\ s_1' \leftarrow \mathsf{tail}\, s_1 \\ s_2' \leftarrow \mathsf{tail}\, s_2 \end{array} \right] . s_1' \sim' s_2'.$$

Using Löb induction we can now obtain a proof

$$\sim^g\text{–to–id} : s_1 \sim^g s_2 \to s_1 = s_2.$$

By using $\blacksquare$ (or clock quantification, as we do in Chapter 3) we can then lift $\sim^g$ to a bisimulation relation $\sim$ on coinductive streams, and lift $\sim^g$–to–id to

$$\sim\text{–to–id} : s_1 \sim s_2 \to s_1 = s_2,$$

and thus we have the coinductive proof principle for streams.

This, of course, is no surprising result as long as our guarded recursive type theory is extensional, since working in an extensional type theory is, in a sense, equivalent to working directly in the model. This provides motivation for designing a guarded recursive type theory with a decidable definitional equality relation.

1.2.9 Path Equalities

In recent years there has been considerable progress on the design of type theories with alternative identity types allowing for some amount of extensionality while retaining decidability of type checking. One such is *cubical*

type theory [29] which bases its identity types on *paths*, a notion from topology and one of the primary objects of study in homotopy theory. The goal of cubical type theory is to provide a computational interpretation of Voevodsky's univalence axiom, and thus it can be considered part of the research on homotopy type theory [84] which combines ideas of type theory and homotopy theory.

In traditional topology a *path* p is a continuous function from the unit interval $[0,1] \subseteq \mathbb{R}$ into a space, and $p(0)$ and $p(1)$ are referred to as the *end-points* of p. In cubical type theory there is a special interval object $\mathbb{I}$ and the following rule for constructing paths in a type A:

$$\frac{\Gamma, i : \mathbb{I} \vdash t : A}{\Gamma \vdash \langle i \rangle\, t : \mathsf{Path}_A\, t[0/i]\; t[1/i]}$$

where 0 and 1 are special elements in $\mathbb{I}$. Similarly we can apply interval elements to paths to obtain a point in A:

$$\frac{\Gamma \vdash t : \mathsf{Path}_A\, u\; v \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash t\, r : A}$$

Since the end-points of a path is encoded in its type, we do not have to inspect the definition of a path to obtain its end-points – this is reflected in the following reduction rules:

$$\begin{aligned} t\, 0 &\mapsto u && \text{if } t \text{ has type } \mathsf{Path}_A\, u\; v \\ t\, 1 &\mapsto v && \text{if } t \text{ has type } \mathsf{Path}_A\, u\; v. \end{aligned}$$

The idea is now that $\mathsf{Path}_A\, u\; v$ replaces the identity type $u =_A v$. In this setting we have a computational interpretation of functional extensionality, witnessed by the following term:

$$\begin{aligned} &\mathsf{funext} : (f\; g : (x : A) \to B) \to ((x : A) \to \mathsf{Path}_B (f\, x)\, (g\, x)) \to \mathsf{Path}_{(x:A)\to B}\, f\; g \\ &\mathsf{funext}\; f\; g\; p = \langle i \rangle\, \lambda x : A.p\, x\, i. \end{aligned}$$

Cubical type theory enjoys canonicity [44] and it is conjectured that it also has decidable type-checking. There is a prototype implementation of a type checker for cubical type theory available at https://github.com/mortberg/cubicaltt.

1.2.10 Guarded Cubical Type Theory

Since cubical type theory provides an identity type with functional extensionality while retaining canonicity and – seemingly – decidable type checking, we decided to experiment with combining cubical type theory with guarded recursive types. In order to retain decidable type-checking we want a decidable

definitional equality relation, and therefore we cannot allow the unrestricted unfolding of fixed-point

$$\mathsf{fix}\, x.t = t[\mathsf{next}\, \mathsf{fix}\, x.t/x].$$

Our solution is to exchange this definitional equality with a path equality, so any fixed-point will be the 0-end-point of a path where the 1-end-point is the unfolding of the fixed point. This, however, can immediately lead to loss of canonicity: $\mathsf{fix}\, x.0$ inhabits Nat, and therefore the term ought not be stuck. To remedy this we simply replace the usual guarded fixed-point combinator $\mathsf{fix} : (\triangleright A \to A) \to A$ with $\mathsf{dfix} : (\triangleright A \to A) \to \triangleright A$, the *delayed* guarded fixed-point combinator which yields a fixed-point which is guarded by a 'later' and therefore does not affect canonicity of base types. The typing rule for dfix is

$$\frac{\Gamma \vdash r : \mathbb{I} \qquad \Gamma, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{dfix}^r\, x.t : \triangleright A}$$

along with the definitional equality

$$\mathsf{dfix}^1\, x.t = \mathsf{next}\, t[\mathsf{dfix}^0\, x.t/x].$$

Now the regular guarded fixed-point combinator can be defined in terms of dfix

$$\mathsf{fix}\, x.t \triangleq t[\mathsf{dfix}^0\, x.t/x]$$

along with a proof of the fixed-point equation:

$$\langle i \rangle\, t[\mathsf{dfix}^i\, x.t/x] : \mathsf{Path}_A(\mathsf{fix}\, x.t)\ (t[\mathsf{next}\, \mathsf{fix}\, x.t/x]).$$

Analogous to how cubical type theory provides a computational interpretation of functional extensionality, cubical type theory with guarded recursive types provides a computational interpretation of the extensionality principle for $\triangleright$:

$$\begin{aligned}&\mathsf{laterext} : \triangleright(\mathsf{Path}_A\ t\ u) \to \mathsf{Path}_{\triangleright A}(\mathsf{next}\, t)\ (\mathsf{next}\, u)\\ &\mathsf{laterext}\ p = \langle i \rangle\, \mathsf{next}\, [p' \leftarrow p].p'\, i.\end{aligned}$$

Chapter 4 is dedicated to the *guarded cubical type theory*, with many examples, and a description of a presheaf model based on both the topos of trees, and the cubical sets model of cubical type theory.

1.2.11 Implementation

The process of developing the guarded cubical type theory involved implementing a prototype type-checker[4] for the language. It is implemented as

[4] https://github.com/hansbugge/cubicaltt/tree/gcubical

a patch on *cubicaltt*[5] which is the type-checker implementation for cubical type theory. All the examples of Chapter 4 are formalised in our prototype. Though absent from the type theory in Chapter 4, our implementation also supports clock quantification and thus coinductive types. This provides the user with the ability of eliminating $\triangleright$'s in a controlled way, however experiments suggest that decidability of type-checking and canonicity holds even with these features.

In order to implement a type checker for a dependently typed language one needs a way of 'running' the terms. This is due to the *conversion* rule:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B}$$

where $A \equiv B$ means that A and B are convertible – e.g., in a setting with reduction semantics, this could mean that $A \to^* C \,{}^*\!\leftarrow B$ for some C. The conversion rule makes sure that definitionally equal types have the same inhabitants. Since types depends on terms, this boils down to a problem of finding an operational semantics for the terms. The most challenging part of implementing the type-checker was indeed to define the operational behaviour of the new terms, especially those involving delayed substitutions. Our solution is to translate delayed substitutions into an alternative representation using 'resource tokens' to remove $\triangleright$'s from the types of certain terms occuring under next and $\triangleright$. In Chapter 5 we discuss the ongoing work of formalising this alternative approach and its reduction semantics.

1.3 Outline of the Dissertation

This dissertation consists of 3 peer-reviewed articles and one chapter with recent preliminary work. Chapter 2 is a extended journal version of an earlier conference paper, while Chapters 3 and 4 are based on conference papers and both include a technical appendix.

1.3.1 Chapter 2: Simple Types

This chapter consists of a journal paper [28] which itself is a considerably extended version of a conference paper [27].

In this paper we present gλ, a simply typed λ-calculus extended with two modalities, $\blacktriangleright$ ('later') and $\blacksquare$ ('everything now'), as well as a fixed-point constructor μ for defining guarded recursive types. The $\blacktriangleright$ and μ originate in Nakano's [69] modal logic for recursion, while $\blacksquare$ is inspired by the *clock quantifiers* of Atkey and McBride [8].

We develop both operational and denotational semantics, using the latter to show normalisation of the former. We show many example programs,

[5] https://github.com/mortberg/cubicaltt

illustrating the expressivity of the new constructs and the difference between programming with guarded recursive types and programming with coinductive types. The operational semantics and the examples are implemented in Agda. The denotational model is the presheaf topos $\mathrm{PSh}(\omega)$ (the *topos of trees*), which we prove is *adequate* with respect to the operational semantics, i.e., that terms that share denotation will be contextually equivalent. The internal language of $\mathrm{PSh}(\omega)$ induces a program logic for gλ which (because of adequacy) can be used to reason about the behaviour of terms.

As a demonstration of the expressivity of gλ we show that we can construct solutions to Rutten's behavioural differential equations [78], which describe a class of coinductive streams.

My contributions to this chapter consist mainly of the design of the type theory and the operational semantics, the Agda implementation of the type theory and the operational semantics, and making examples.

1.3.2 Chapter 3: Dependent Types

This chapter consist of a conference paper [22] along with a technical appendix which provides more detailed examples.

In this paper we present *guarded dependent type theory* (GDTT), an extensional dependent type theory with guarded recursive types and clock quantification. Like in the simple type theory of the previous chapter there is a 'later' modality (now symbolised with the more discreet $\triangleright$), however the rules for the simply typed version do not immediately generalise to a dependently typed setting. Specifically, there is a problem with the interaction with function application: the result type of a function application *depends* on the argument term – so if the argument term is delayed by a 'later' then the result type cannot be defined. The solution is to enrich the modality with *delayed substitutions*, as mentioned in the introduction, which allows for the formation of such result types. Delayed substitutions, which occur both on type and term levels, turn out to be a powerful programming construct.

Instead of the 'everything now' modality of gλ, GDTT has Atkey and McBride [8] style clock quantifiers, which means that the 'later's are now decorated with *clock variables*, and we can universally quantify over these clocks to obtain coinductive types.

Another difference from the simply typed gλ is that GDTT does not have a μ for defining guarded recursive types. It instead has a guarded fixed-point combinator for *terms*, and with this guarded recursive types can be defined by taking fixed points on the universe, an idea due to Birkedal and Møgelberg [10].

We show in detail examples of programming with guarded recursive and coinductive types, and by taking fixed points of identity types we prove properties about programs within the language itself.

The rules of GDTT are justified by a denotational model by Bizjak and Møgelberg [18].

It is important to note that GDTT is an *extensional* type theory, i.e., it contains the equality reflection rule. Therefore type checking is undecidable.

For this chapter I contributed to the design of the typing rules, in particular the delayed substitutions, and to designing examples.

1.3.3 Chapter 4: Cubical Types

This chapter consists of a conference paper [15] along with a technical appendix which provides details of the model construction.

In this paper we tackle the issue of decidable type checking of dependent type theory with guarded recursive types. Our approach is to combine *cubical type theory* [29] (CTT) with guarded recursive types à la GDTT. From CTT we get the *path equality* type, which replaces the extensional identity type of GDTT, thus we have an equality type which provides a computational interpretation of functional extensionality, which turns out to also provide a computational interpretation of an extensionality principle for 'later' types. This principle is crucial for reasoning about guarded recursive data. To control the unfolding of guarded fixed-points we have made the fixed-point equation into a path equality. The resulting type theory is called *guarded cubical type theory* (GCTT). We conjecture that GCTT enjoys both decidable type checking and canonicity.

There is a prototype implementation of a type checker for GCTT based on an existing type checker for CTT which provides confidence in the syntactic properties of the theory.

The semantics of GCTT is given via the presheaf category over the product of the categories used to define GDTT and CTT.

Clock quantification is a feature of GDTT which is lacking in this presentation of GCTT. This is due to the inevitable complexity of a model with multiple clocks and clock synchronisation, which would be put on top of the already intricate model of the present paper. However, the prototype type checker already supports most of the rules for clock quantification, and thus coinductive types.

I contributed to the design of the type theory and the construction of the model. Furthermore I implemented the type checker along with Andrea Vezzosi, and designed and formalised the examples.

1.3.4 Chapter 5: Reduction Semantics

This chapter contains a discussion of currently ongoing and future work on reduction semantics for a version of guarded recursive type theory without identity types. We describe the idea of guarded recursive types with *resources*, which is an alternative to using delayed substitutions that allows us to for-

mulate a reduction relation on terms and types. Once we have a reduction relation we can formulate properties like confluence, subject reduction, and strong normalisation, all of which we conjecture to hold.

This chapter is based on joint work with Patrick Bahr and Rasmus E. Møgelberg.

1.4 Concluding Remarks

Our motivation for investigating guarded recursive type theory were twofold.

(i) Guarded recursive types as a tool for programming and reasoning with coinductive types.

(ii) Guarded recursive type theory as a foundation for formalising models of programming languages.

In this section I will discuss the current status with respect to these two points.

Programming and reasoning with coinductive types. In Chapter 3 we have described a type theory in which we have successfully encoded proofs and programs of coinductive types. But this cannot rightly be considered 'programming' for at least one reason: the lack of operational semantics. In Chapter 5 there is preliminary work on a reduction relation for a dependently typed language without identity types, and the implementation of guarded cubical type theory has provided experimental evidence that an operational semantics is conceivable. Unfortunately, we have not yet found a way to justify operationally the equality rule TmEq-∀-fresh of Figure 3.6, which seems to be necessary for programming with dependent coinductive types such as covectors (see Section 3.5). The problems arising from this equality rule seem to be related to parametricity problems.

The guarded cubical type theory of Chapter 4 does not involve coinductive types. It is our conjecture, based on experiments with the implementation, that we can extend the type theory with clock quantification (without the TmEq-∀-fresh rule) while retaining canonicity and decidable type-checking. The soundness of such a type theory has not yet been shown. We conjecture that it is possible to combine the current model of guarded cubical type theory with the techniques for modelling the clocks of the extensional guarded recursive type theory, however there are significant gaps to fill out. Such a type theory would be a good witness for the usefulness of guarded recursion for proving and programming with coinductive types in one language.

Formalising models. Paviotti et al. [72] and Møgelberg and Paviotti [68] have recently used the type theory from Chapter 3 informally to model the programming languages PCF and FPC. This gives us confidence that guarded

recursive type theory can be a useful tool for making models of programming languages which would otherwise be done using step-indexing. However, our current implementation of guarded cubical type theory is still in its infancy, and this hinders experimentation with formalisations of such models. Lacking features, such as hidden arguments and automation, makes large formalisation efforts very tedious. When modelling programming languages in Agda one would typically utilise inductive families, which have not yet been combined with guarded recursive types.

The landscape of theorem provers based on dependent type theory is currently changing, in part because of the developments of homotopy type theory. Once a more mature theorem prover with a computational interpretation of functional extensionality emerges, this would be an obvious place to begin larger scale experiments with guarded recursive types.

Chapter 2

Simple Types

This chapter is a version of the paper:

[28] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal.
The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types.
Logical Methods of Computer Science (LMCS), 12(3), 2016.

which is a journal version of a conference paper:

[27] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal.
Programming and Reasoning with Guarded Recursion for Coinductive Types.
In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 407–421, 2015.

The only difference with the published version is that a few typos has been fixed.

Abstract

We present the guarded lambda-calculus, an extension of the simply typed lambda-calculus with guarded recursive and coinductive types. The use of guarded recursive types ensures the productivity of well-typed programs. Guarded recursive types may be transformed into coinductive types by a type-former inspired by modal logic and Atkey-McBride clock quantification, allowing the typing of acausal functions. We give a call-by-name operational semantics for the calculus, and define adequate denotational semantics in the topos of trees. The adequacy proof entails that the evaluation of a program always terminates. We introduce a program logic with Löb induction for reasoning about the contextual equivalence of programs. We demonstrate the expressiveness of the calculus by showing the definability of solutions to Rutten's behavioural differential equations.

2.1 Introduction

The problem of ensuring that functions on coinductive types are well-defined has prompted a wide variety of work into productivity checking, and rule formats for coalgebra. *Guarded recursion* [31] guarantees unique solutions for definitions, as well as their *productivity* – any finite prefix of the solution can be produced in finite time by unfolding – by requiring that recursive calls on a coinductive data type be nested under its constructor; for example, cons (written ::) for streams. This can sometimes be established by a simple syntactic check, as for the stream toggle and binary stream function interleave below:

```
toggle = 1 :: 0 :: toggle
interleave (x :: xs) ys = x :: interleave ys xs
```

Such syntactic checks, however, exclude many valid definitions in the presence of higher order functions. For example, consider the *regular paperfolding sequence* (also, more colourfully, known as the *dragon curve sequence* [83]), which describes the sequence of left and right folds induced by repeatedly folding a piece of paper in the same direction. This sequence, with left and right folds encoded as 1 and 0, can be defined via the function interleave as follows [39]:

```
paperfolds = interleave toggle paperfolds
```

This definition is productive, but the putative definition below, which also applies interleave to two streams and so should apparently have the same type, is not:

```
paperfolds' = interleave paperfolds' toggle
```

This equation is satisfied by any stream whose *tail* is the regular paperfolding sequence, so lacks a unique solution. Unfortunately syntactic productivity checking, such as that employed by the proof assistant Coq [63], will fail to detect the difference between these programs, and reject both.

A more flexible approach, first suggested by Nakano [69], is to guarantee productivity via *types*. A new modality, for which we follow Appel et al. [7] by writing $\blacktriangleright$ and using the name 'later', allows us to distinguish between data we have access to now, and data which we have only later. This $\blacktriangleright$ must be used to guard self-reference in type definitions, so for example *guarded streams* over the natural numbers $\mathbf{N}$ are defined by the guarded recursive equation

$$\mathsf{Str}^{\mathsf{g}}\,\mathbf{N} \triangleq \mathbf{N} \times \blacktriangleright \mathsf{Str}^{\mathsf{g}}\,\mathbf{N}$$

asserting that stream heads are available now, but tails only later. The type of interleave will be $\mathsf{Str}^{\mathsf{g}}\,\mathbf{N} \to \blacktriangleright \mathsf{Str}^{\mathsf{g}}\,\mathbf{N} \to \mathsf{Str}^{\mathsf{g}}\,\mathbf{N}$, capturing the fact the (head of the) first argument is needed immediately, but the second argument is needed

only later. In term definitions the types of self-references will then be guarded by ▶ also. For example interleave paperfolds′ toggle becomes ill-formed, as the paperfolds′ self-reference has type ▶Str^{g} **N**, rather than Str^{g} **N** as required, but interleave toggle paperfolds will be well-formed.

Adding ▶ alone to the simply typed λ-calculus enforces a discipline more rigid than productivity. For example the obviously productive stream function

```
every2nd (x :: x' :: xs) = x :: every2nd xs
```

cannot be typed because it violates *causality* [56]: elements of the result stream depend on deeper elements of the argument stream. In some settings, such as functional reactive programming, this is a desirable property, but for productivity guarantees alone it is too restrictive – we need the ability to remove ▶ in a controlled way. This is provided by the *clock quantifiers* of Atkey and McBride [8], which assert that all data is available now. This does not trivialise the guardedness requirements because there are side-conditions restricting how clock quantifiers may be introduced. Moreover clock quantifiers allow us to recover first-class *coinductive* types from guarded recursive types, while retaining our productivity guarantees.

Note on this point that our presentation departs from Atkey and McBride's [8] by regarding the 'everything now' operator as a unary type-former, written ■ and called 'constant', rather than a quantifier. Observing that the types $\blacksquare A \to A$ and $\blacksquare A \to \blacksquare\blacksquare A$ are always inhabited allows us to see this type-former, via the Curry-Howard isomorphism, as an *S4* modality, and hence base this part of our calculus on the established typed calculi for intuitionistic S4 (IS4) of Bierman and de Paiva [9]. We will discuss the trade-offs involved in this alternative presentation in our discussion of related work in Section 2.6.1.

Overview of our contributions. In Section 2.2 we present the guarded λ-calculus, more briefly referred to as the gλ-calculus, extending the simply typed λ-calculus with guarded recursive and coinductive types. We define call-by-name operational semantics, which will prevent the indefinite unfolding of recursive functions, an obvious source of non-termination. In Section 2.3 we define denotational semantics in the topos of trees [14] which are *adequate*, in the sense that denotationally equal terms behave identically in any context, and as a corollary to the logical relations argument used to establish adequacy, prove normalisation of the calculus.

We are interested not only in *programming* with guarded recursive and coinductive types, but also in *proving* properties of these programs; in Section 2.4 we show how the internal logic of the topos of trees induces the program logic L gλ for reasoning about the denotations of gλ-programs. Given the adequacy of our semantics, this logic permits proofs about the operational behaviour of terms. In Section 2.5 we demonstrate the expressiveness of

the gλ-calculus by showing the definability of solutions to Rutten's behavioural differential equations [78], and show that $L\mathrm{g}\lambda$ can be used to reason about them, as an alternative to standard bisimulation-based arguments. In Section 2.6 we conclude with a discussion of related and further work.

This paper is based on a previously published conference paper [27], but has been significantly revised and extended. We have improved the presentation of our results and examples throughout the paper, but draw particular attention to the following changes:

- We present in the body of this paper many proof details that previously appeared only in an appendix to the technical report version of the conference paper [26].
- We discuss sums, and in particular the interaction between sums and the constant modality via the box^+ term-former, which previously appeared only in an appendix to the technical report. We further improve on that discussion by presenting conatural numbers as a motivating example; by giving new equational rules for box^+ in Section 2.4.2; and by proving a property of box^+ in Section 2.4.3.
- We present new examples in Example 2.11 which show that converting a program to type-check in the gλ-calculus is not always straightforward.
- We give a more intuitive introduction to the logic $L\mathrm{g}\lambda$ in Section 2.4, aimed at readers who are not experts in topos theory. In particular we see how the guarded conatural numbers define the type of propositions.
- We present new equational rules in Section 2.4.2 that reveal how the explicit substitutions of the gλ-calculus interact with real substitutions.
- We present (slightly improved) results regarding total and inhabited types in the gλ-calculus in Section 2.4.2 which previously appeared only in an appendix to the technical report. Relatedly, we have generalised the proof in Example 2.42.1 to remove its requirement that the type in question is total and inhabited, by including a new equational rule regarding composition for applicative functors.
- We present formal results regarding behavioural differential equations in Section 2.5 which previously appeared only in an appendix to the technical report.
- We conduct a much expanded discussion of related and further work in Section 2.6.

We have implemented the gλ-calculus in Agda, a process we found helpful when fine-tuning the design of our calculus. The implementation, with many examples, is available online.[1]

2.2 The Guarded Lambda-Calculus

This section presents the guarded λ-calculus, more briefly referred to as the gλ-calculus, its call-by-name operational semantics, and its types, then gives some examples.

2.2.1 Untyped Terms and Operational Semantics

In this subsection we will see the untyped gλ-calculus and its call-by-name operational semantics. This calculus takes the usual λ-calculus with natural numbers, products, coproducts, and (iso-)recursion, and makes two extensions. First, the characteristic operations of *applicative functors* [64], here called next and $\circledast$, are added, which will support the definition of causal guarded recursive functions. Second, a prev (previous) term-former is added, inverse to next, that along with box and unbox term-formers will support the definition of acausal functions without sacrificing guarantees of productivity.

The novel term-formers of the gλ-calculus are most naturally understood as operations on its novel types. We will therefore postpone any examples of gλ-calculus terms until after we have seen its types.

Note that we will later add one more term-former, called box^+, to allow us to write more programs involving the interaction of binary sums and the box term-former. We postpone discussion of this term-former until Section 2.2.4 to allow a cleaner presentation of the core system.

Definition 2.1. *Untyped* gλ*-terms* are defined by the grammar

$$
\begin{array}{rcll}
t & ::= & x & \text{(variables)} \\
 & | & \mathsf{zero} \mid \mathsf{succ}\, t & \text{(natural numbers)} \\
 & | & \langle\rangle \mid \langle t, t\rangle \mid \pi_1 t \mid \pi_2 t & \text{(products)} \\
 & | & \mathsf{abort}\, t \mid \mathsf{in}_1\, t \mid \mathsf{in}_2\, t \mid \mathsf{case}\, t\, \mathsf{of}\, x_1.t; x_2.t & \text{(sums)} \\
 & | & \lambda x.t \mid tt & \text{(functions)} \\
 & | & \mathsf{fold}\, t \mid \mathsf{unfold}\, t & \text{(recursion operations)} \\
 & | & \mathsf{next}\, t \mid \mathsf{prev}\, \sigma.t \mid t \circledast t & \text{(‘later’ operations)} \\
 & | & \mathsf{box}\, \sigma.t \mid \mathsf{unbox}\, t & \text{(‘constant’ operations)}
\end{array}
$$

where σ is an *explicit substitution*: a list of variables and terms $[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$, often abbreviated as $[\vec{x} \leftarrow \vec{t}]$. We write $\mathsf{prev}\, \iota.t$ for $\mathsf{prev}[\vec{x} \leftarrow \vec{x}].t$, where $\vec{x}$ is a list of all free variables of t, and write $\mathsf{prev}\, t$ where $\vec{x}$ is empty. We similarly write $\mathsf{box}\, \iota.t$ and $\mathsf{box}\, t$.

[1] http://users-cs.au.dk/hbugge/bin/glambda.zip

The terms $\mathsf{prev}[\vec{x} \leftarrow \vec{t}].t$ and $\mathsf{box}[\vec{x} \leftarrow \vec{t}].t$ bind all variables of $\vec{x}$ in t, but *not* in $\vec{t}$. We adopt the convention that prev and box have highest precedence.

Definition 2.2. The *reduction rules* on closed gλ-terms are

$$\begin{array}{rcll}
\pi_d\langle t_1, t_2\rangle & \mapsto & t_d & (d \in \{1,2\}) \\
\mathsf{case}\,\mathsf{in}_d\, t\,\mathsf{of}\, x_1.t_1; x_2.t_2 & \mapsto & t_d[t/x_d] & (d \in \{1,2\}) \\
(\lambda x.t_1)t_2 & \mapsto & t_1[t_2/x] & \\
\mathsf{unfold}\,\mathsf{fold}\,t & \mapsto & t & \\
\mathsf{prev}[\vec{x} \leftarrow \vec{t}].t & \mapsto & \mathsf{prev}(t[\vec{t}/\vec{x}]) & (\vec{x}\ \textit{non-empty}) \\
\mathsf{prev}\,\mathsf{next}\,t & \mapsto & t & \\
\mathsf{next}\,t_1 \circledast \mathsf{next}\,t_2 & \mapsto & \mathsf{next}(t_1\,t_2) & \\
\mathsf{unbox}(\mathsf{box}[\vec{x} \leftarrow \vec{t}].t) & \mapsto & t[\vec{t}/\vec{x}] &
\end{array}$$

All rules above except that concerning $\circledast$ look like standard β-reduction, removing 'roundabouts' of introduction then elimination. A partial exception to this observation are the prev and next rules; an apparently more conventional β-rule for these term-formers would be

$$\mathsf{prev}[\vec{x} \leftarrow \vec{t}].(\mathsf{next}\,t) \mapsto t[\vec{t}/\vec{x}] \tag{2.1}$$

Where $\vec{x}$ is non-empty this rule might require us to reduce an *open* term to derive $\mathsf{next}\,t$, for the computation to continue. But it is, as usual, easy to construct examples of open terms that get stuck without reducing to a value, even where they are well-typed (by the rules of the next subsection). Therefore a closed well-typed term of form $\mathsf{prev}[\vec{x} \leftarrow \vec{t}].u$ may not see u reduce to some $\mathsf{next}\,u'$, and so if equation (2.1) were the only applicable rule the term as a whole would also be stuck.

This is not necessarily a problem for us, because we are not interested in unrestricted reduction. Such reduction is not compatible in a total calculus with the presence of infinite structures such as streams, as we could choose to unfold a stream indefinitely and hence normalisation would be lost. In this paper we will instead adopt a strategy where we prohibit the reduction of open terms; specifically we will use call-by-name evaluation. In the case above we manage this by first applying the explicit substitution without eliminating prev.

The rule involving $\circledast$ is not a true β-rule, as $\circledast$ is neither introduction nor elimination, but is necessary to enable function application under a next and hence allow, for example, manipulation of the tail of a stream. It corresponds to the 'homomorphism' equality for applicative functors [64].

We next impose our call-by-name strategy on these reductions.

Definition 2.3. *Values* are terms of the form

$$\mathsf{succ}^n\,\mathsf{zero} \mid \langle\rangle \mid \langle t, t\rangle \mid \mathsf{in}_1\,t \mid \mathsf{in}_2\,t \mid \lambda x.t \mid \mathsf{fold}\,t \mid \mathsf{next}\,t \mid \mathsf{box}\,\sigma.t$$

where succ^n is a list of zero or more succ operators, and t is any term.

Definition 2.4. *Evaluation contexts* are defined by the grammar

$$\begin{array}{rcl} E & ::= & \cdot \mid \mathsf{succ}\, E \mid \pi_1 E \mid \pi_2 E \mid \mathsf{case}\, E\, \mathsf{of}\, x_1.t_1; x_2.t_2 \mid Et \mid \mathsf{unfold}\, E \\ & \mid & \mathsf{prev}\, E \mid E \circledast t \mid v \circledast E \mid \mathsf{unbox}\, E \end{array}$$

If we regard $\circledast$ naively as function application, it is surprising in a call-by-name setting that its right-hand side may be reduced. However both sides must be reduced until they have main connective next, before the reduction rule for $\circledast$ may be applied. Thus the order of reductions of gλ-terms cannot be identified with the order of the call-by-name reductions of the corresponding λ-calculus term with the novel connectives erased.

Definition 2.5. *Call-by-name reduction* has format $E[t] \mapsto E[u]$, where $t \mapsto u$ is a reduction rule. From now the symbol $\mapsto$ will be reserved to refer to call-by-name reduction. We use $\rightsquigarrow$ for the reflexive transitive closure of $\mapsto$.

Note that the call-by-name reduction relation $\mapsto$ is deterministic.

2.2.2 Types

We now meet the typing rules of the gλ-calculus, the most important feature of which is the restriction of the fixed point constructor μ to *guarded* occurrences of recursion variables.

Definition 2.6. Open gλ-*types* are defined by the grammar

$$\begin{array}{rcll} A & ::= & \alpha & \text{(type variables)} \\ & \mid & \mathbf{N} & \text{(natural numbers)} \\ & \mid & \mathbf{1} \mid A \times A & \text{(products)} \\ & \mid & \mathbf{0} \mid A + A & \text{(sums)} \\ & \mid & A \to A & \text{(functions)} \\ & \mid & \mu\alpha.A & \text{(iso-recursive types)} \\ & \mid & \blacktriangleright A & \text{(later)} \\ & \mid & \blacksquare A & \text{(constant)} \end{array}$$

Type formation rules are defined inductively by the rules of Figure 2.1. In this figure ∇ is a finite set of type variables, and a variable α is *guarded in* a type A if all occurrences of α are beneath an occurrence of $\blacktriangleright$ in the syntax tree. We adopt the convention that unary type-formers bind closer than binary type-formers. All types in this paper will be understood as closed unless explicitly stated otherwise.

Note that the guardedness side-condition on the μ type-former and the prohibition on the formation of $\blacksquare A$ for open A together create a prohibition on applying $\mu\alpha$ to any α with $\blacksquare$ above it, for example $\mu\alpha.\blacksquare\blacktriangleright\alpha$ or $\mu\alpha.\blacktriangleright\blacksquare\alpha$. This accords with our intuition that fixed points will exist only where a recursion variable is 'displaced in time' by a $\blacktriangleright$. The constant type-former $\blacksquare$ destroys any such displacement by giving 'everything now'.

$$\frac{}{\nabla \vdash \alpha}\ \alpha \in \nabla \qquad \frac{}{\nabla \vdash \mathbf{N}} \qquad \frac{}{\nabla \vdash \mathbf{1}} \qquad \frac{\nabla \vdash A_1 \qquad \nabla \vdash A_2}{\nabla \vdash A_1 \times A_2} \qquad \frac{}{\nabla \vdash \mathbf{0}}$$

$$\frac{\nabla \vdash A_1 \qquad \nabla \vdash A_2}{\nabla \vdash A_1 + A_2} \qquad \frac{\nabla \vdash A_1 \qquad \nabla \vdash A_2}{\nabla \vdash A_1 \to A_2} \qquad \frac{\nabla, \alpha \vdash A}{\nabla \vdash \mu\alpha.A}\ \alpha \,\mathsf{guarded\,in}\, A$$

$$\frac{\nabla \vdash A}{\nabla \vdash \blacktriangleright A} \qquad \frac{\cdot \vdash A}{\nabla \vdash \blacksquare A}$$

Figure 2.1: Type formation for the gλ-calculus

Definition 2.7. The *typing judgments* are given in Figure 2.2. There Γ is a *typing context*, i.e. a finite set of variables x, each associated with a type A, written $x : A$. In the side-conditions to the prev and box rules, types are *constant* if all occurrences of $\blacktriangleright$ are beneath an occurrence of $\blacksquare$ in their syntax tree.

The *constant* types exist 'all at once', due to the absence of $\blacktriangleright$ or presence of $\blacksquare$; this condition corresponds to the freeness of the clock variable in Atkey and McBride [8] (recalling that this paper's work corresponds to the use of only one clock). Its use as a side-condition to $\blacksquare$-introduction in Figure 2.2 recalls (but is more general than) the 'essentially modal' condition in the natural deduction calculus of Prawitz [76] for the modal logic Intuitionistic S4 (IS4). The term calculus for IS4 of Bierman and de Paiva [9], on which this calculus is most closely based, uses the still more restrictive requirement that $\blacksquare$ be the main connective. This would preclude some functions that seem desirable, such as the isomorphism $\lambda n.\,\mathsf{box}\,\iota.n : \mathbf{N} \to \blacksquare\mathbf{N}$.

The presence of explicit substitutions attached to the prev and box can seem heavy notationally, but in practice the burden on the programmer seems quite small, as in all examples we will see, prev appears only in its syntactic sugar forms

$$\frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : \blacktriangleright A}{\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash \mathsf{prev}\,\iota.t : A}\ A_1, \ldots, A_n \,\mathsf{constant} \qquad \frac{\cdot \vdash t : \blacktriangleright A}{\Gamma \vdash \mathsf{prev}\,t : A}$$

and similarly for box. One might therefore ask why the more general form involving explicit substitutions is necessary. The answer is that the 'sugared' definitions above are not closed under substitution: we need $(\mathsf{prev}\,\iota.t)[\vec{u}/\vec{x}] = \mathsf{prev}[\vec{x} \leftarrow \vec{u}].t$. In general getting substitution right in the presence of side-conditions can be rather delicate. The solution we use, namely *closing* the term t to which prev (or box) is applied to protect its variables, comes directly from Bierman and de Paiva's calculus for IS4 [9]; see this reference for more

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathsf{zero} : \mathbf{N}} \qquad \frac{\Gamma \vdash t : \mathbf{N}}{\Gamma \vdash \mathsf{succ}\, t : \mathbf{N}} \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}}$$

$$\frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1 t : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2 t : B} \qquad \frac{\Gamma \vdash t : \mathbf{0}}{\Gamma \vdash \mathsf{abort}\, t : A}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{in}_1\, t : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{in}_2\, t : A + B}$$

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma, x_1 : A \vdash t_1 : C \qquad \Gamma, x_2 : B \vdash t_2 : C}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, x_1.t_1; x_2.t_2 : C} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \qquad \frac{\Gamma \vdash t : A[\mu\alpha.A/\alpha]}{\Gamma \vdash \mathsf{fold}\, t : \mu\alpha.A} \qquad \frac{\Gamma \vdash t : \mu\alpha.A}{\Gamma \vdash \mathsf{unfold}\, t : A[\mu\alpha.A/\alpha]}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \blacktriangleright A} \qquad \frac{\begin{array}{c} x_1 : A_1, \ldots, x_n : A_n \vdash t : \blacktriangleright A \\ \Gamma \vdash t_1 : A_1 \qquad \cdots \qquad \Gamma \vdash t_n : A_n \end{array}}{\Gamma \vdash \mathsf{prev}[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n].t : A}\ A_1, \ldots, A_n\ \mathsf{constant}$$

$$\frac{\Gamma \vdash t_1 : \blacktriangleright(A \to B) \qquad \Gamma \vdash t_2 : \blacktriangleright A}{\Gamma \vdash t_1 \circledast t_2 : \blacktriangleright B}$$

$$\frac{\begin{array}{c} x_1 : A_1, \ldots, x_n : A_n \vdash t : A \\ \Gamma \vdash t_1 : A_1 \qquad \cdots \qquad \Gamma \vdash t_n : A_n \end{array}}{\Gamma \vdash \mathsf{box}[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n].t : \blacksquare A}\ A_1, \ldots, A_n\ \mathsf{constant} \qquad \frac{\Gamma \vdash t : \blacksquare A}{\Gamma \vdash \mathsf{unbox}\, t : A}$$

Figure 2.2: Typing rules for the gλ-calculus

in-depth discussion of the issue, and in particular how a failure to account for this issue causes problems for the calculus of Prawitz [76]. Similar side-conditions have also caused problems in the closely related area of calculi with clocks – see the identification by Bizjak and Møgelberg [19] of a problem with the type theory presented in earlier work by Møgelberg [66].

Lemma 2.8 (Subject Reduction for Closed Terms)**.** *$\vdash t : A$ and $t \rightsquigarrow u$ implies $\vdash u : A$.* □

Note that the reduction rule

$$\mathsf{prev}[\vec{x} \leftarrow \vec{t}].t \mapsto \mathsf{prev}(t[\vec{t}/\vec{x}])$$

plainly violates subject reduction for open terms: the right hand side is only well-defined if $t[\vec{t}/\vec{x}]$ has no free variables, because the explicit substitution attached to prev must close all open variables.

2.2.3 Examples

We may now present example gλ-programs and their typings. We will first give causal programs without use of the constant modality $\blacksquare$, then show how this modality expands the expressivity of the language, and finally show two examples of productive functions which are a bit trickier to fit within our language.

Example 2.9.

1. The type of guarded recursive streams over some type A, written $\mathsf{Str}^{\mathsf{g}}A$, is, as noted in the introduction, defined as $\mu\alpha.A \times \blacktriangleright\alpha$. Other guarded recursive types can be defined, such as infinite binary trees as $\mu\alpha.A \times \blacktriangleright(\alpha \times \alpha)$, conatural numbers $\mathsf{CoNat}^{\mathsf{g}}$ as $\mu\alpha.1 + \blacktriangleright\alpha$, and colists as $\mu\alpha.1 + (A \times \blacktriangleright\alpha)$. We will focus on streams in this section, and look more at $\mathsf{CoNat}^{\mathsf{g}}$ in Section 2.2.4.

2. We define guarded versions of the standard stream functions cons (written infix as ::), head, and tail as obvious:

$$\begin{array}{rcll} :: & \triangleq & \lambda x.\lambda s.\,\mathsf{fold}\langle x, s\rangle & : \quad A \to \blacktriangleright\mathsf{Str}^{\mathsf{g}}A \to \mathsf{Str}^{\mathsf{g}}A \\ \mathsf{hd}^{\mathsf{g}} & \triangleq & \lambda s.\pi_1\,\mathsf{unfold}\,s & : \quad \mathsf{Str}^{\mathsf{g}}A \to A \\ \mathsf{tl}^{\mathsf{g}} & \triangleq & \lambda s.\pi_2\,\mathsf{unfold}\,s & : \quad \mathsf{Str}^{\mathsf{g}}A \to \blacktriangleright\mathsf{Str}^{\mathsf{g}}A \end{array}$$

   We can then use the $\circledast$ term-former to make observations deeper into the stream:

$$\begin{array}{rcll} \mathsf{2nd}^{\mathsf{g}} & \triangleq & \lambda s.(\mathsf{next}\,\mathsf{hd}^{\mathsf{g}}) \circledast (\mathsf{tl}^{\mathsf{g}}\,s) & : \quad \mathsf{Str}^{\mathsf{g}}A \to \blacktriangleright A \\ \mathsf{3rd}^{\mathsf{g}} & \triangleq & \lambda s.(\mathsf{next}\,\mathsf{2nd}^{\mathsf{g}}) \circledast (\mathsf{tl}^{\mathsf{g}}\,s) & : \quad \mathsf{Str}^{\mathsf{g}}A \to \blacktriangleright\blacktriangleright A \cdots \end{array}$$

3. To define guarded recursive functions we need a fixed point combinator. Abel and Vezzosi [3] gave a guarded version of Curry's Y combinator in a similar calculus; for variety we present a version of Turing's fixed point combinator.

   Recall from the standard construction that if we had a μ type-former with no guardedness requirements, then a combinator fix with type $(A \to A) \to A$ could be defined, for any type A, by the following:

$$\begin{array}{rcll} \mathsf{Rec}_A & \triangleq & \mu\alpha.(\alpha \to (A \to A) \to A) & \\ \theta & \triangleq & \lambda y.\lambda f.f((\mathsf{unfold}\,y)yf) & : \quad \mathsf{Rec}_A \to (A \to A) \to A \\ \mathsf{fix} & \triangleq & \theta(\mathsf{fold}\,\theta) & : \quad (A \to A) \to A \end{array}$$

To see that fix does indeed behave as a fixpoint, note that $\mathsf{fix}\, f$ unfolds in one step to $f((\mathsf{unfold}\,\mathsf{fold}\,\theta)(\mathsf{fold}\,\theta)f)$. But $\mathsf{unfold}\,\mathsf{fold}$ eliminates[2], so we have $f(\mathsf{fix}\, f)$.

What then is the guarded version of this combinator? Following the need for the recursion variable to be guarded, and the original observation of Nakano [69] that guarded fixed point combinators should have type $(\blacktriangleright A \to A) \to A$, we reconstruct the type Rec_A by the addition of later modalities in the appropriate places. The terms θ and fix can then be constructed by adding next term-formers, and replacing function application with $\circledast$, to the original terms so that they type-check:

$$\begin{array}{rcl} \mathsf{Rec}_A & \triangleq & \mu\alpha.(\blacktriangleright\alpha \to (\blacktriangleright A \to A) \to A) \\ \theta & \triangleq & \lambda y.\lambda f.f((\mathsf{next}\,\lambda z.\,\mathsf{unfold}\,z) \circledast y \circledast \mathsf{next}\, y \circledast \mathsf{next}\, f) : \\ & & \blacktriangleright \mathsf{Rec}_A \to (\blacktriangleright A \to A) \to A \\ \mathsf{fix} & \triangleq & \theta(\mathsf{next}\,\mathsf{fold}\,\theta) : (\blacktriangleright A \to A) \to A \end{array}$$

The addition of these novel term-formers is fairly mechanical; the only awkward point comes when we cannot unfold y directly because it has type $\blacktriangleright \mathsf{Rec}_A$ rather than Rec_A, so we must introduce the expression $\lambda z.\,\mathsf{unfold}\,z$.

Now $\mathsf{fix}\, f$ reduces to

$$f((\mathsf{next}\,\lambda z.\,\mathsf{unfold}\,z) \circledast (\mathsf{next}\,\mathsf{fold}\,\theta) \circledast (\mathsf{next}\,\mathsf{next}\,\mathsf{fold}\,\theta) \circledast \mathsf{next}\, f)$$

But the reduction rule for $\circledast$ allows us to take next out the front and replace $\circledast$ by normal application:

$$f(\mathsf{next}((\lambda z.\,\mathsf{unfold}\,z)(\mathsf{fold}\,\theta)(\mathsf{next}\,\mathsf{fold}\,\theta)f))$$

Applying the λ-expression and eliminating $\mathsf{unfold}\,\mathsf{fold}$ yields $f(\mathsf{next}\,\mathsf{fix}\, f)$. In other words, we have defined a standard fixed point except that a next is added to the term to record that the next application of the fixed point combinator must take place one step in the future. We will be able to be more formal about this property of fix in Lemma 2.40, once we have introduced the program logic $L\mathrm{g}\lambda$ for reasoning about $\mathrm{g}\lambda$-programs.

Note that the inhabited type $(\blacktriangleright A \to A) \to A$ does not imply that all types are inhabited, as there is not in general a function $\blacktriangleright A \to A$. This differs from the standard presentation of fixed point combinators that leads to inconsistency.

4. Given our fixed point combinator we may now build some guarded streams; for example, the simple program (in pseudocode)

[2] With respect to call-by-name evaluation this program's next reduction will depend on the shape of f, but it is enough for this discussion to see that $\mathsf{unfold}\,\mathsf{fold}\,\theta$ is equal to θ in the underlying equational theory.

```
zeros = 0 :: zeros
```

is captured by the term

$$\mathsf{zeros} \triangleq \mathsf{fix}\,\lambda s.(\mathsf{zero} :: s)$$

of type $\mathsf{Str}^{\mathsf{g}}\mathbf{N}$. Here s has type $\blacktriangleright\mathsf{Str}^{\mathsf{g}}\mathbf{N}$, and so the function that the fixed point is applied to has type $\blacktriangleright\mathsf{Str}^{\mathsf{g}}\mathbf{N} \to \mathsf{Str}^{\mathsf{g}}\mathbf{N}$; exactly the type expected by fix.

Note however that the plainly unproductive stream definition

```
circular = circular
```

cannot be defined within this calculus, although it is it apparently definable via a standard fixed point combinator as $\mathsf{fix}\,\lambda s.s$; in our calculus the type of the recursion variable s must be preceded by a $\blacktriangleright$ modality.

5. For a slightly more sophisticated example, consider the standard map function on streams:

$$\mathsf{map}^{\mathsf{g}} \triangleq \lambda f.\,\mathsf{fix}\,\lambda m.\lambda s.(f\ \mathsf{hd}^{\mathsf{g}}\,s) :: (m \circledast \mathsf{tl}^{\mathsf{g}}\,s) : (A \to B) \to \mathsf{Str}^{\mathsf{g}}A \to \mathsf{Str}^{\mathsf{g}}B$$

Here the recursion variable m has type $\blacktriangleright(\mathsf{Str}^{\mathsf{g}}A \to \mathsf{Str}^{\mathsf{g}}B)$.

6. We can define two more standard stream functions – iterate, which takes a function $A \to A$ and a head A, and produces a stream by applying the function repeatedly, and interleave, which interleaves two streams – in the obvious ways:

$$\begin{array}{rcl} \mathsf{iterate}' & \triangleq & \lambda f.\,\mathsf{fix}\,\lambda g.\lambda x.x :: (g \circledast \mathsf{next}(f\,x)) \\ & : & (A \to A) \to A \to \mathsf{Str}^{\mathsf{g}}A \end{array}$$

$$\begin{array}{rcl} \mathsf{interleave}' & \triangleq & \mathsf{fix}\,\lambda g.\lambda s.\lambda t.(\mathsf{hd}^{\mathsf{g}}\,s) :: (g \circledast (\mathsf{next}\,t) \circledast \mathsf{tl}^{\mathsf{g}}\,s) \\ & : & \mathsf{Str}^{\mathsf{g}}A \to \mathsf{Str}^{\mathsf{g}}A \to \mathsf{Str}^{\mathsf{g}}A \end{array}$$

These definitions are correct but are less informative than they could be, as they do not record the temporal aspects of these functions, namely that (in the case of iterate) the function, and (in the case of interleave) the second stream, are not used until the next time step. We could alternatively use the definitions

$$\begin{array}{rcl} \mathsf{iterate} & \triangleq & \lambda f.\,\mathsf{fix}\,\lambda g.\lambda x.x :: (g \circledast (f \circledast \mathsf{next}\,x)) \\ & : & \blacktriangleright(A \to A) \to A \to \mathsf{Str}^{\mathsf{g}}A \end{array}$$

$$\begin{array}{rcl} \mathsf{interleave} & \triangleq & \mathsf{fix}\,\lambda g.\lambda s.\lambda t.(\mathsf{hd}^{\mathsf{g}}\,s) :: (g \circledast t \circledast \mathsf{next}\,\mathsf{tl}^{\mathsf{g}}\,s) \\ & : & \mathsf{Str}^{\mathsf{g}}A \to \blacktriangleright\mathsf{Str}^{\mathsf{g}}A \to \mathsf{Str}^{\mathsf{g}}A \end{array}$$

These definitions are in fact more general:

$$\begin{array}{rcl} \mathsf{iterate}'\,f\,x & = & \mathsf{iterate}(\mathsf{next}\,f)\,x \\ \mathsf{interleave}'\,s\,t & = & \mathsf{interleave}\,s\,(\mathsf{next}\,t) \end{array}$$

Indeed the example of the regular paperfolding sequence from the introduction shows that the more general and informative version can also be more useful:

$$\begin{array}{rcll} \mathsf{toggle} & \triangleq & \mathsf{fix}\,\lambda s.(\mathsf{succ}\,\mathsf{zero}) :: (\mathsf{next}(\mathsf{zero} :: s)) & : & \mathsf{Str}^{\mathsf{g}}\,\mathbf{N} \\ \mathsf{paperfolds} & \triangleq & \mathsf{fix}\,\lambda s.\,\mathsf{interleave}\,\mathsf{toggle}\,s & : & \mathsf{Str}^{\mathsf{g}}\,\mathbf{N} \end{array}$$

The recursion variable s in paperfolds has type $\blacktriangleright\mathsf{Str}^{\mathsf{g}}\,\mathbf{N}$, which means it cannot be given as the second argument to interleave′ – only the more general interleave will do. However the erroneous definition of the regular paperfolding sequence that replaced $\mathsf{interleave}\,\mathsf{toggle}\,s$ with $\mathsf{interleave}'\,s$ toggle cannot be typed.

Another example of a function that (rightly) cannot be typed in $g\lambda$ is a filter function on streams which eliminates elements that fail some boolean test; as all elements may fail the test, the function is not productive.

7. μ-types define *unique* fixed points, carrying both initial algebra and final coalgebra structure. For example, the type $\mathsf{Str}^{\mathsf{g}}A$ is both the initial algebra and the final coalgebra for the functor $A \times \blacktriangleright$-. This contrasts with the usual case of streams, which are merely the final coalgebra for the functor $A \times$ -; the initial algebra for this functor is trivial. To see the dual structure of guarded recursive types, consider the functions[3]

$$\begin{array}{rcl} \mathsf{initial} & \triangleq & \mathsf{fix}\,\lambda g.\lambda f.\lambda s.f\langle \mathsf{hd}^{\mathsf{g}}\,s, g \circledast \mathsf{next}\,f \circledast \mathsf{tl}^{\mathsf{g}}\,s\rangle \\ & : & ((A \times \blacktriangleright B) \to B) \to \mathsf{Str}^{\mathsf{g}}A \to B \end{array}$$

$$\begin{array}{rcl} \mathsf{final} & \triangleq & \mathsf{fix}\,\lambda g.\lambda f.\lambda x.(\pi_1(f\,x)) :: (g \circledast \mathsf{next}\,f \circledast \pi_2(f\,x)) \\ & : & (B \to A \times \blacktriangleright B) \to B \to \mathsf{Str}^{\mathsf{g}}A \end{array}$$

For example, $\mathsf{map}^{\mathsf{g}}\,h : \mathsf{Str}^{\mathsf{g}}A \to \mathsf{Str}^{\mathsf{g}}A$ can be written as $\mathsf{initial}\,\lambda x.(h(\pi_1 x)) :: (\pi_2 x)$, or as $\mathsf{final}\,\lambda s.\langle h(\mathsf{hd}^{\mathsf{g}}\,s), \mathsf{tl}^{\mathsf{g}}\,s\rangle$.

We now turn to examples involving the prev (previous) term-former and constant modality $\blacksquare$.

Example 2.10.

1. The $\blacksquare$ type-former lifts guarded recursive streams to coinductive streams, as we will make precise in Example 2.16. We define $\mathsf{Str}A \triangleq \blacksquare\mathsf{Str}^{\mathsf{g}}A$. We can then define versions of cons, head, and tail operators for coinductive streams:

$$\begin{array}{rcll} \mathsf{cons} & \triangleq & \lambda x.\lambda s.\,\mathsf{box}\,\iota.x :: \mathsf{next}(\mathsf{unbox}\,s) & : & A \to \mathsf{Str}A \to \mathsf{Str}A \\ \mathsf{hd} & \triangleq & \lambda s.\,\mathsf{hd}^{\mathsf{g}}(\mathsf{unbox}\,s) & : & \mathsf{Str}A \to A \\ \mathsf{tl} & \triangleq & \lambda s.\,\mathsf{box}\,\iota.\,\mathsf{prev}\,\iota.\,\mathsf{tl}^{\mathsf{g}}(\mathsf{unbox}\,s) & : & \mathsf{Str}A \to \mathsf{Str}A \end{array}$$

[3] These are usually called fold and unfold; we avoid this because of the name clash with our term-formers.

Note that cons is well-defined only if A is a constant type. Note also that we must 'unbox' our coinductive stream to turn it into a guarded stream before we operate on it. This explains why we retain our productivity guarantees. Finally, note the absence of $\blacktriangleright$ in the types. Indeed we can define observations deeper into the stream with no hint of later, for example

$$\mathsf{2nd} \triangleq \lambda s.\, \mathsf{hd}(\mathsf{tl}\, s) : \mathsf{Str} A \to A$$

2. We have a general way to lift boxed functions to functions on boxed types, via the 'limit' function

$$\mathsf{lim} \triangleq \lambda f.\lambda x.\, \mathsf{box}\, \iota.(\mathsf{unbox}\, f)(\mathsf{unbox}\, x) : \blacksquare(A \to B) \to \blacksquare A \to \blacksquare B$$

This allows us to lift our guarded stream functions from Example 2.9 to coinductive stream functions, provided that the function in question is defined in a constant environment. For example

$$\mathsf{map} \triangleq \lambda f.\, \mathsf{lim}\, \mathsf{box}\, \iota.(\mathsf{map}^{\mathsf{g}}\, f) : (A \to B) \to \mathsf{Str} A \to \mathsf{Str} B$$

is definable if $A \to B$ is a constant type (which is to say, A and B are constant types).

3. The more sophisticated acausal function $\mathsf{every2nd} : \mathsf{Str} A \to \mathsf{Str}^{\mathsf{g}} A$ is

$$\mathsf{fix}\, \lambda g.\lambda s.(\mathsf{hd}\, s) :: (g \circledast \mathsf{next}(\mathsf{tl}(\mathsf{tl}\, s)))$$

Note that it takes a *coinductive* stream $\mathsf{Str} A$ as argument. The function with coinductive result type is then $\lambda s.\, \mathsf{box}\, \iota.\, \mathsf{every2nd}\, s : \mathsf{Str} A \to \mathsf{Str} A$.

4. Guarded streams do not define a monad, as the standard 'diagonal' join function $\mathsf{Str}^{\mathsf{g}}(\mathsf{Str}^{\mathsf{g}} A) \to \mathsf{Str}^{\mathsf{g}} A$ cannot be defined, as for example the second element of the second stream in $\mathsf{Str}^{\mathsf{g}}(\mathsf{Str}^{\mathsf{g}} A)$ has type $\blacktriangleright\blacktriangleright A$, while the second element of the result stream should have type $\blacktriangleright A$ – the same problem as for every2nd above. However we can define

$$\mathsf{diag} \triangleq \mathsf{fix}\, \lambda f.(\mathsf{hd}(\mathsf{hd}\, s)) :: (f \circledast \mathsf{next}(\mathsf{tl}(\mathsf{tl}\, s))) : \mathsf{Str}(\mathsf{Str} A) \to \mathsf{Str}^{\mathsf{g}} A$$

The standard join function is then $\lambda s.\, \mathsf{box}\, \iota.\mathsf{diag}\, s : \mathsf{Str}(\mathsf{Str} A) \to \mathsf{Str} A$.

In the examples above the construction of typed gλ-terms from the standard definitions of productive functions required little ingenuity; one merely applies the new type- and term-formers in the 'necessary places' until everything type-checks. This appears to be the case with the vast majority of such functions. However, below are two counter-examples, both from Endrullis et al. [40], where a bit more thought is required:

Example 2.11.

1. The *Thue-Morse sequence* is a stream of booleans which can be defined (in pseudo-code) as

   ```
   thuemorse  = 0 :: tl (h thuemorse)
   h (0 :: s) = 0 :: 1 :: (h s)
   h (1 :: s) = 1 :: 0 :: (h s)
   ```

   The definition of thuemorse is productive only because the helper stream function h produces two elements of its result stream after reading one element of its input stream. To see that this is crucial, observe that if we replace h by the identity stream function, thuemorse is no longer productive. The type of h therefore needs to be something other than $\mathrm{Str}^g(\mathbf{1}+\mathbf{1}) \to \mathrm{Str}^g(\mathbf{1}+\mathbf{1})$. But it does not have type $\blacktriangleright\mathrm{Str}^g(\mathbf{1}+\mathbf{1}) \to \mathrm{Str}^g(\mathbf{1}+\mathbf{1})$ because it needs to read the head of its input stream before it produces the first element of its output stream. Capturing this situation – a stream function that produces nothing at step zero, but two elements at step one – seems too fine-grained to fit well with our calculus with $\blacktriangleright$.

   The simplest solution is to modify the definition above by unfolding the definition of thuemorse once:

   ```
   thuemorse = 0 :: 1 :: h (tl (h thuemorse))
   ```

   This equivalent definition *would* remain productive if we replaced h with the identity, and so h can be typed $\mathrm{Str}^g(\mathbf{1}+\mathbf{1}) \to \mathrm{Str}^g(\mathbf{1}+\mathbf{1})$ without problem.

2. The definition below of the *Fibonacci word* is similar to the example above, but shows that the situation can be even more intricate:

   ```
   fibonacci  = 0 :: tl (f fibonacci)
   f (0 :: s) = 0 :: 1 :: (f s)
   f (1 :: s) = 0 :: (f s)
   ```

   Here the helper function f, if given a stream with head 0, produces nothing at step zero, but two elements at step one, as for h above. But given a stream with head 1, it produces only one element at step one. Therefore the erroneous definition

   ```
   fibonacci' = 1 :: tl (f fibonacci')
   ```

   whose head is 1 rather than 0, is not productive. Productivity hence depends on an inspection of *terms*, rather than merely types, in a manner clearly beyond the scope of our current work.

   Again, this can be fixed by unfolding the definition once:

   ```
   fibonacci = 0 :: 1 :: f (tl (f fibonacci))
   ```

2.2.4 Sums and the Constant Modality

Atkey and McBride's calculus with clocks [8] includes as a primitive notion *type equalities* regarding the interaction of clock quantification with other type-formers. They note that most of these equalities are not essential, as in many cases mutually inverse terms between the sides of the equalities are definable. However this is not so with, among other cases, binary sums. Binary sums present a similar problem for our calculus. We can define a term

$$\lambda x.\,\mathsf{box}\,\iota.\,\mathsf{case}\,x\,\mathsf{of}\,x_1.\,\mathsf{in}_1\,\mathsf{unbox}\,x_1\,;x_2.\,\mathsf{in}_2\,\mathsf{unbox}\,x_2 : (\blacksquare A + \blacksquare B) \to \blacksquare(A+B)$$

in our calculus but no term in general in the other direction. Unfortunately such a term is essential to defining some basic operations involving coinductive types involving sums. For example we define the (guarded and coinductive) conatural numbers as

$$\begin{array}{rcl} \mathsf{CoNat}^{\mathsf{g}} & \triangleq & \mu\alpha.(1+\blacktriangleright\alpha) \\ \mathsf{CoNat} & \triangleq & \blacksquare\mathsf{CoNat}^{\mathsf{g}} \end{array}$$

These correspond to natural numbers with infinity, with such programs definable upon them as

$$\begin{array}{rcll} \mathsf{cozero} & \triangleq & \mathsf{fold}(\mathsf{in}_1\,\langle\rangle) & : \quad \mathsf{CoNat}^{\mathsf{g}} \\ \mathsf{cosucc} & \triangleq & \lambda n.\,\mathsf{fold}(\mathsf{in}_2(\mathsf{next}\,n)) & : \quad \mathsf{CoNat}^{\mathsf{g}} \to \mathsf{CoNat}^{\mathsf{g}} \\ \mathsf{infinity} & \triangleq & \mathsf{fix}\,\lambda n.\,\mathsf{fold}(\mathsf{in}_2\,n) & : \quad \mathsf{CoNat}^{\mathsf{g}} \end{array}$$

As a guarded recursive construction, $\mathsf{CoNat}^{\mathsf{g}}$ defines a unique fixed point. In particular its coalgebra map $\mathsf{pred}^{\mathsf{g}}$ (for 'predecessor') is simply

$$\mathsf{pred}^{\mathsf{g}} \triangleq \lambda n.\,\mathsf{unfold}\,n : \mathsf{CoNat}^{\mathsf{g}} \to 1 + \blacktriangleright\mathsf{CoNat}^{\mathsf{g}}$$

Now the coinductive type CoNat should be a coalgebra also, so we should be able to define a function $\mathsf{pred} : \mathsf{CoNat} \to 1 + \mathsf{CoNat}$ similarly. However a term of type CoNat must be unboxed before it is unfolded, and the type $1 + \blacktriangleright\mathsf{CoNat}^{\mathsf{g}}$ that results is not constant, and so we cannot apply prev and box to map from $\blacktriangleright\mathsf{CoNat}^{\mathsf{g}}$ to CoNat.

Our solution is to introduce a new term-former box^+ which will allow us to define a term

$$\lambda x.\,\mathsf{box}^+\,\iota.\,\mathsf{unbox}\,x : \blacksquare(A+B) \to \blacksquare A + \blacksquare B$$

Definition 2.12 (ref. Definitions 2.1, 2.2, 2.4, 2.7)**.** We extend the grammar of gλ*-terms* by

$$t \quad ::= \quad \cdots \mid \mathsf{box}^+\,\sigma.t$$

where σ is an explicit substitution. We abbreviate terms with box^+ as for prev and box.

We extend the reduction rules with

$$\begin{array}{rcll} \mathsf{box}^+[\vec{x} \leftarrow \vec{t}].t & \mapsto & \mathsf{box}^+\, t[\vec{t}/\vec{x}] & (\vec{x}\ \textit{non-empty}) \\ \mathsf{box}^+\, \mathsf{in}_d\, t & \mapsto & \mathsf{in}_d\, \mathsf{box}\, t & (d \in \{1,2\}) \end{array}$$

We do not change the definition of values of Definition 2.3. We extend the definition of evaluation contexts with

$$E \quad ::= \quad \cdots \mid \mathsf{box}^+\, E$$

Finally, we add the new typing judgment

$$\frac{\begin{array}{c} x_1 : A_1, \ldots, x_n : A_n \vdash t : B_1 + B_2 \\ \Gamma \vdash t_1 : A_1 \qquad \cdots \qquad \Gamma \vdash t_n : A_n \end{array}}{\Gamma \vdash \mathsf{box}^+[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n].t : \blacksquare B_1 + \blacksquare B_2}\ A_1, \ldots, A_n\ \mathsf{constant}$$

Returning to our example, we can define the term $\mathsf{pred} : \mathsf{CoNat} \to 1 + \mathsf{CoNat}$ as

$$\lambda n.\, \mathsf{case}(\mathsf{box}^+\, \iota.\, \mathsf{unfold}\, \mathsf{unbox}\, n)\, \mathsf{of}\, x_1.\, \mathsf{in}_1\, \langle\rangle; x_2.\, \mathsf{in}_2\, \mathsf{box}\, \iota.\, \mathsf{prev}\, \iota.\, \mathsf{unbox}\, x_2$$

2.3 Denotational Semantics and Normalisation

This section gives denotational semantics for gλ-types and terms, as objects and arrows in the topos of trees [14], the presheaf category over the first infinite ordinal $\omega \triangleq 1 \leq 2 \leq \cdots$[4] (we give a concrete definition below). The denotational semantics are shown to be sound and, by a logical relations argument, adequate with respect to the operational semantics. Normalisation follows as a corollary of this argument.

2.3.1 The Topos of Trees

This section introduces the mathematical model in which our denotational semantics will be defined.

Definition 2.13. The *topos of trees* $\mathcal{S}$ has, as objects X, families of sets $X_1, X_2, \ldots$ indexed by the positive integers, equipped with families of *restriction functions* $r_i^X : X_{i+1} \to X_i$ indexed similarly. Arrows $f : X \to Y$ are families of functions $f_i : X_i \to Y_i$ indexed similarly obeying the *naturality* condition $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$:

$$\begin{array}{ccccccc} X_1 & \xleftarrow{r_1^X} & X_2 & \xleftarrow{r_2^X} & X_3 & \xleftarrow{r_3^X} & \cdots \\ \downarrow{\scriptstyle f_1} & & \downarrow{\scriptstyle f_2} & & \downarrow{\scriptstyle f_3} & & \\ Y_1 & \xleftarrow[r_1^Y]{} & Y_2 & \xleftarrow[r_2^Y]{} & Y_3 & \xleftarrow[r_3^Y]{} & \cdots \end{array}$$

[4] It would be more standard to start this pre-order at 0, but we start at 1 to maintain harmony with some equivalent presentations of the topos of trees and related categories which have a vacuous stage 0; we shall see such a presentation in Section 2.5.3

Given an object X and positive integers $i \leq j$ we write $\restriction_i$ for the function $X_j \to X_i$ defined by composing the restriction functions r_k^X for $k \in \{i, i+1, \ldots, j-1\}$, or as the identity where $i = j$.

$\mathcal{S}$ is a cartesian closed category with products and coproducts defined pointwise. Note that by naturality it holds that for any arrow $f : X \to Y + Z$, positive integer n, and element $x \in X_n$, $f_i \circ \restriction_i(x)$ must be an element of the same side of the sum for all $i \leq n$. The exponential A^B has, as its component sets $(A^B)_i$, the set of i-tuples $(f_1 : A_1 \to B_1, \ldots, f_i : A_i \to B_i)$ obeying the naturality condition, and projections as restriction functions.

Definition 2.14.

1. The category of sets **Set** is a full subcategory of $\mathcal{S}$ via the functor Δ : **Set** $\to \mathcal{S}$ that maps sets Z to the $\mathcal{S}$-object

$$Z \xleftarrow{id_Z} Z \xleftarrow{id_Z} Z \xleftarrow{id_Z} \cdots$$

   and maps functions f by $(\Delta f)_i = f$ similarly.

   The full subcategory of *constant objects* consists of $\mathcal{S}$-objects which are *isomorphic* to objects of the form ΔZ. These are precisely the objects whose restriction functions are bijections. In particular the terminal object 1 of $\mathcal{S}$ is $\Delta\{*\}$, the initial object is $\Delta\emptyset$, and the *natural numbers object* is $\Delta\mathbb{N}$;

   We will abuse notation slightly and treat constant objects as if they were actually of the form ΔZ, i.e., if X is constant and $x \in X_i$ we will write x also, for example, for the element $\left(r_i^X\right)^{-1}(x) \in X_{i+1}$.

2. Δ is left adjoint to the 'global elements' functor $hom_{\mathcal{S}}(1, -)$. We write $\blacksquare$ for the endofunctor $\Delta \circ hom_{\mathcal{S}}(1, \text{-}) : \mathcal{S} \to \mathcal{S}$. Then $\mathsf{unbox} : \blacksquare \dot{\to} id_{\mathcal{S}}$ is the counit of the comonad associated with this adjunction. Concretely, for any $\mathcal{S}$-object X and $x \in hom_{\mathcal{S}}(1, X)$ we have $\mathsf{unbox}_i(x) = x_i$, i.e. the i'th component of $x : 1 \to X$ applied to the unique element $*$:

$$\begin{array}{ccccccc}
hom_{\mathcal{S}}(1,X) & \xleftarrow{id} & hom_{\mathcal{S}}(1,X) & \xleftarrow{id} & hom_{\mathcal{S}}(1,X) & \xleftarrow{id} & \cdots \\
{\scriptstyle x \mapsto x_1}\downarrow & & {\scriptstyle x \mapsto x_2}\downarrow & & {\scriptstyle x \mapsto x_3}\downarrow & & \\
X_1 & \xleftarrow[r_1^X]{} & X_2 & \xleftarrow[r_2^X]{} & X_3 & \xleftarrow[r_3^X]{} & \cdots
\end{array}$$

   The global elements functor can also be understood by considering an $\mathcal{S}$-object X as a diagram in **Set**; then $hom_{\mathcal{S}}(1, X)$ is its *limit*, and so $\blacksquare X$ is this limit considered as a $\mathcal{S}$-object.

3. $\blacktriangleright : \mathcal{S} \to \mathcal{S}$ is defined by mapping $\mathcal{S}$-objects X to

$$\{*\} \xleftarrow{!} X_1 \xleftarrow{r_1^X} X_2 \xleftarrow{r_2^X} \cdots$$

That is, $(\blacktriangleright X)_1 = \{*\}$ and $(\blacktriangleright X)_{i+1} = X_i$, with $r_1^{\blacktriangleright X}$ defined uniquely and $r_{i+1}^{\blacktriangleright X} = r_i^X$. The $\blacktriangleright$ functor acts on arrows $f : X \to Y$ by $(\blacktriangleright f)_1 = id_{\{*\}}$ and $(\blacktriangleright f)_{i+1} = f_i$. The natural transformation $\mathsf{next} : id_{\mathcal{S}} \dot{\to} \blacktriangleright$ has, for each component X, next_1 uniquely defined and $\mathsf{next}_{i+1} = r_i^X$:

$$\begin{array}{ccccccccc}
X_1 & \xleftarrow{r_1^X} & X_2 & \xleftarrow{r_2^X} & X_3 & \xleftarrow{r_3^X} & \cdots \\
\downarrow{\scriptstyle !} & & \downarrow{\scriptstyle r_1^X} & & \downarrow{\scriptstyle r_2^X} & & \\
\{*\} & \xleftarrow[!]{} & X_1 & \xleftarrow[r_1^X]{} & X_2 & \xleftarrow[r_2^X]{} & \cdots
\end{array}$$

2.3.2 Denotational Semantics

We may now see how the gλ-calculus can be interpreted soundly in the topos of trees.

Definition 2.15. We interpret types in context $\nabla \vdash A$, where ∇ contains n free variables, as functors $[\![\nabla \vdash A]\!] : (\mathcal{S}^{op} \times \mathcal{S})^n \to \mathcal{S}$, usually written $[\![A]\!]$. This mixed variance definition is necessary as variables may appear negatively or positively.

- $[\![\nabla, \alpha \vdash \alpha]\!]$ is the projection of the objects or arrows corresponding to *positive* occurrences of α, e.g. $[\![\alpha]\!](\vec{W}, X, Y) = Y$;
- $[\![\mathbf{N}]\!]$, $[\![\mathbf{1}]\!]$, and $[\![\mathbf{0}]\!]$ are the constant functors $\Delta\mathbb{N}$, $\Delta\{*\}$, and $\Delta\emptyset$ respectively;
- $[\![A_1 \times A_2]\!](\vec{W}) = [\![A_1]\!](\vec{W}) \times [\![A_2]\!](\vec{W})$. The definition of the functor on $\mathcal{S}$-arrows is likewise pointwise;
- $[\![A_1 + A_2]\!](\vec{W}) = [\![A_1]\!](\vec{W}) + [\![A_2]\!](\vec{W})$ similarly;
- $[\![\mu\alpha.A]\!](\vec{W}) = \mathsf{Fix}(F)$, where $F : (\mathcal{S}^{op} \times \mathcal{S}) \to \mathcal{S}$ is the functor given by $F(X, Y) = [\![A]\!](\vec{W}, X, Y)$ and $\mathsf{Fix}(F)$ is the unique (up to isomorphism) X such that $F(X, X) \cong X$. The existence of such X relies on F being a suitably locally contractive functor, which follows by Birkedal et al. [14, Section 4.5] and the fact that $\blacksquare$ is only ever applied to closed types. This restriction on $\blacksquare$ is necessary because the functor $\blacksquare$ is not *strong*.
- $[\![A_1 \to A_2]\!](\vec{W}) = [\![A_2]\!](\vec{W})^{[\![A_2]\!](\vec{W}')}$ where $\vec{W}'$ is $\vec{W}$ with odd and even elements switched to reflect change in polarity, i.e. $(X_1, Y_1, \ldots)' = (Y_1, X_1, \ldots)$;
- $[\![\blacktriangleright A]\!], [\![\blacksquare A]\!]$ are defined by composition with the functors $\blacktriangleright, \blacksquare$ (Def. 2.14).

Example 2.16.

1. $[\![\mathrm{Str}^{\mathrm{g}}\,\mathbf{N}]\!]$ is the $\mathcal{S}$-object

$$\mathbb{N} \xleftarrow{pr_1} \mathbb{N}\times\mathbb{N} \xleftarrow{pr_1} (\mathbb{N}\times\mathbb{N})\times\mathbb{N} \xleftarrow{pr_1} \cdots$$

where the pr_1 are first projection functions. This is intuitively the object of *approximations* of streams – first the head, then the first two elements, and so forth. Conversely, $[\![\mathrm{Str}\,\mathbf{N}]\!] = \Delta(\mathbb{N}^\omega)$, so it is the constant object of streams, as usually defined in **Set**. This can also be understood as the limit of the approximations given by $[\![\mathrm{Str}^{\mathrm{g}}\,\mathbf{N}]\!]$.

More generally, any polynomial functor F on **Set** can be assigned a gλ-type A_F with a free type variable α that occurs guarded. The denotation of $\blacksquare\mu\alpha.A_F$ will then be the constant object of the carrier of the final coalgebra for F [66, Theorem 2]. Therefore $\blacksquare$ is the modality that takes us from guarded recursive constructions to coinductive constructions.

2. $[\![\mathsf{CoNat}^{\mathrm{g}}]\!]$ is the $\mathcal{S}$-object

$$2 \xleftarrow{r_1^\Omega} 3 \xleftarrow{r_2^\Omega} 4 \xleftarrow{r_3^\Omega} \cdots$$

where each set n is $\{0,1,\ldots,n-1\}$ and $r_n^\Omega(k) = \min(n,k)$. In fact this is the *subobject classifier* of $\mathcal{S}$, usually written Ω.

$[\![\mathsf{CoNat}]\!]$ is the constant object $\Delta(\mathbb{N}+\{\infty\})$.

Lemma 2.17. *The interpretation of a recursive type is isomorphic to the interpretation of its unfolding:* $[\![\mu\alpha.A]\!](\vec{W}) \cong [\![A[\mu\alpha.A/\alpha]]\!](\vec{W})$. □

Lemma 2.18. *Constant types denote constant objects in* $\mathcal{S}$.

Proof. By induction on type formation, with $\blacktriangleright A$ case omitted, $\blacksquare A$ a base case, and $\mu\alpha.A$ considered only where α is not free in A. □

Note that the converse does not apply; for example $[\![\blacktriangleright 1]\!]$ is a constant object.

Definition 2.19. We interpret typing contexts $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ in the usual way as $\mathcal{S}$-objects $[\![\Gamma]\!] \triangleq [\![A_1]\!]\times\cdots\times[\![A_n]\!]$, and hence interpret typed terms-in-context $\Gamma \vdash t : A$ as $\mathcal{S}$-arrows $[\![\Gamma \vdash t : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$ (usually written $[\![t]\!]$) as follows.

$[\![x]\!]$ is the projection $[\![\Gamma]\!]\times[\![A]\!] \to [\![A]\!]$. $[\![\mathsf{zero}]\!]$ and $[\![\mathsf{succ}\,t]\!]$ are as obvious. Term-formers for products and function spaces are interpreted via the cartesian closed structure of $\mathcal{S}$, and for sums via its coproducts. Exponentials are not merely pointwise, so we give the definitions explicitly:

- $[\![\lambda x.t]\!]_i(\gamma)_j$ maps $a \mapsto [\![\Gamma, x : A \vdash t : B]\!]_j(\restriction_j(\gamma), a)$;
- $[\![t_1\,t_2]\!]_i(\gamma) = ([\![t_1]\!]_i(\gamma)_i) \circ [\![t_2]\!]_i(\gamma)$;

$[\![\mathsf{fold}\, t]\!]$ and $[\![\mathsf{unfold}\, t]\!]$ are defined via composition with the isomorphisms of Lemma 2.17. $[\![\mathsf{next}\, t]\!]$ and $[\![\mathsf{unbox}\, t]\!]$ are defined by composition with the natural transformations introduced in Definition 2.14. The final cases are

- $[\![t_1 \circledast t_2]\!]_1$ is defined uniquely at the trivial first stage of the denotation of a later type; $[\![t_1 \circledast t_2]\!]_{i+1}(\gamma) \triangleq ([\![t_1]\!]_{i+1}(\gamma)_i) \circ [\![t_2]\!]_{i+1}(\gamma)$.
- $[\![\mathsf{prev}[x_1 \leftarrow t_1, \ldots].t]\!]_i(\gamma) \triangleq [\![t]\!]_{i+1}([\![t_1]\!]_i(\gamma), \ldots)$, where $[\![t_1]\!]_i(\gamma) \in [\![A_1]\!]_i$ is also in $[\![A_1]\!]_{i+1}$ by Lemma 2.18;
- $[\![\mathsf{box}[x_1 \leftarrow t_1, \ldots].t]\!]_i(\gamma)_j = [\![t]\!]_j([\![t_1]\!]_i(\gamma), \ldots)$, again using Lemma 2.18;
- Let $[\![t]\!]_j([\![t_1]\!]_i(\gamma), \ldots, [\![t_n]\!]_i(\gamma))$ (which is well-defined by Lemma 2.18) be $[a_j, d]$ as j ranges, recalling that $d \in \{1, 2\}$ is the same for all i by naturality. Define a to be the arrow $1 \to [\![A_d]\!]$ that has j'th element a_j. Then $\left[\!\left[\mathsf{box}^+[\vec{x} \leftarrow \vec{t}].t\right]\!\right]_i(\gamma) \triangleq [a, d]$.

Lemma 2.20. *Take typed terms in context $x_1 : A_1, \ldots, x_m : A_m \vdash t : A$ and $\Gamma \vdash t_k : A_k$ for all $1 \le k \le m$. Then $\left[\!\left[t[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma) = [\![t]\!]_i([\![t_1]\!]_i(\gamma), \ldots, [\![t_m]\!]_i(\gamma))$.*

Proof. By induction on the typing of t. We present the cases particular to our calculus.

$\mathsf{next}\, t$: case $i = 1$ is trivial.

$$\begin{aligned}
\left[\!\left[\mathsf{next}\, t[\vec{t}/\vec{x}]\right]\!\right]_{i+1}(\gamma) &= r_i^{[\![A]\!]} \circ \left[\!\left[t[\vec{t}/\vec{x}]\right]\!\right]_{i+1}(\gamma) && \text{by definition}\\
&= r_i^{[\![A]\!]} \circ [\![t]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots) && \text{by induction}\\
&= [\![\mathsf{next}\, t]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots).
\end{aligned}$$

$$\begin{aligned}
\left[\!\left[(\mathsf{prev}[\vec{y} \leftarrow \vec{u}].t)[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma) &= \left[\!\left[\mathsf{prev}[\vec{y} \leftarrow \vec{u}[\vec{t}/\vec{x}]].t\right]\!\right]_i(\gamma)\\
&= [\![t]\!]_{i+1}(\left[\!\left[u_1[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma), \ldots) && \text{by definition}\\
&= [\![t]\!]_{i+1}([\![u_1]\!]_i([\![t_1]\!]_i(\gamma), \ldots), \ldots) && \text{by induction}\\
&= [\![\mathsf{prev}[\vec{y} \leftarrow \vec{u}].t]\!]_i([\![t_1]\!]_i(\gamma), \ldots).
\end{aligned}$$

$u_1 \circledast u_2$: case $i = 1$ is trivial.

$$\begin{aligned}
\left[\!\left[(u_1 \circledast u_2)[\vec{t}/\vec{x}]\right]\!\right]_{i+1}(\gamma) &= (\left[\!\left[u_1[\vec{t}/\vec{x}]\right]\!\right]_{i+1}(\gamma)_i) \circ \left[\!\left[u_2[\vec{t}/\vec{x}]\right]\!\right]_{i+1}(\gamma)\\
&= ([\![u_1]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots)_i) \circ [\![u_2]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots)\\
&= [\![u_1 \circledast u_2]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots).
\end{aligned}$$

$$\begin{aligned}
\left[\!\left[\mathsf{box}[\vec{y} \leftarrow \vec{u}[\vec{t}/\vec{x}]].t\right]\!\right]_i(\gamma)_j &= [\![t]\!]_j(\left[\!\left[u_1[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma), \ldots)\\
&= [\![t]\!]_j([\![u_1]\!]_i([\![t_1]\!]_i(\gamma), \ldots), \ldots) && \text{by induction}\\
&= [\![\mathsf{box}[\vec{y} \leftarrow \vec{u}].t]\!]_i([\![t_1]\!]_i(\gamma), \ldots)_j.
\end{aligned}$$

$$\begin{aligned}\left[\!\left[\mathsf{unbox}\, t[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma) &= \left[\!\left[t[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma)_i \\ &= [\![t]\!]_i([\![t_1]\!]_i(\gamma),\ldots)_i \\ &= [\![\mathsf{unbox}\, t]\!]_i([\![t_1]\!]_i(\gamma),\ldots).\end{aligned}$$

$\mathsf{box}^+[\vec{y} \leftarrow \vec{u}].t$: By induction we have $\left[\!\left[u_k[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma) = [\![u_k]\!]_i([\![t_1]\!]_i(\gamma),\ldots)$. Hence $[\![t]\!]_j(\left[\!\left[u_1[\vec{t}/\vec{x}]\right]\!\right]_i(\gamma),\ldots) = [\![t]\!]_j([\![u_1]\!]_i([\![t_1]\!]_i(\gamma),\ldots),\ldots)$ as required. □

Theorem 2.21 (Soundness)**.** *If $t \rightsquigarrow u$ then $[\![t]\!] = [\![u]\!]$.*

Proof. We verify the reduction rules of Definition 2.2; extending this to any evaluation context, and to $\rightsquigarrow$, is easy. The product reduction case is standard, and function case requires Lemma 2.20. unfoldfold is the application of mutually inverse arrows.

$\left[\!\left[\mathsf{prev}[\vec{x} \leftarrow \vec{t}].t\right]\!\right]_i = [\![t]\!]_{i+1}([\![t_1]\!]_i,\ldots)$. Each t_k in the explicit substitution is closed, so is denoted by an arrow from 1 to a constant $\mathcal{S}$-object, so by naturality $[\![t_k]\!]_i = [\![t_k]\!]_{i+1}$. $[\![t]\!]_{i+1}([\![t_1]\!]_{i+1},\ldots) = \left[\!\left[t[\vec{t}/\vec{x}]\right]\!\right]_{i+1}$ by Lemma 2.20, which is $\left[\!\left[\mathsf{prev}\, t[\vec{t}/\vec{x}]\right]\!\right]_i$.

$[\![\mathsf{prev}\,\mathsf{next}\, t]\!]_i = [\![\mathsf{next}\, t]\!]_{i+1} = [\![t]\!]_i$.

With $\circledast$-reduction, index 1 is trivial. $[\![\mathsf{next}\, t_1 \circledast \mathsf{next}\, t_2]\!]_{i+1} = ([\![\mathsf{next}\, t_1]\!]_{i+1})_i \circ [\![\mathsf{next}\, t_2]\!]_{i+1} = (r_i^{[\![A\to B]\!]} \circ [\![t_1]\!]_{i+1})_i \circ r_i^{[\![A]\!]} \circ [\![t_2]\!]_{i+1} = ([\![t_1]\!]_i \circ r_i^1)_i \circ [\![t_2]\!]_i \circ r_i^1$ by naturality, which is $([\![t_1]\!]_i)_i \circ [\![t_2]\!]_i = [\![t_1 t_2]\!]_i = [\![t_1 t_2]\!]_i \circ r_i^1 = r_i^{[\![B]\!]} \circ [\![t_1 t_2]\!]_{i+1} = [\![\mathsf{next}(t_1 t_2)]\!]_{i+1}$.

$\left[\!\left[\mathsf{unbox}(\mathsf{box}[\vec{x} \leftarrow \vec{t}].t)\right]\!\right]_i = (\left[\!\left[\mathsf{box}[\vec{x} \leftarrow \vec{t}].t\right]\!\right]_i)_i = [\![t]\!]_i([\![t_1]\!]_i,\ldots) = \left[\!\left[t[\vec{t}/\vec{x}]\right]\!\right]_i$.

box^+-reduction: Because each $[\![A_k]\!]$ is a constant object (Lemma 2.18), $[\![t_k]\!]_i = [\![t_k]\!]_j$ for all i, j. Hence $\left[\!\left[\mathsf{box}^+[\vec{x} \leftarrow \vec{t}].t\right]\!\right]_i$ is defined via components $[\![t]\!]_j([\![t_1]\!]_j,\ldots)$ and $\left[\!\left[\mathsf{box}^+\, t[\vec{t}/\vec{x}]\right]\!\right]$ is defined via components $\left[\!\left[t[\vec{t}/\vec{x}]\right]\!\right]_j$. These are equal by Lemma 2.20. $[\![\mathsf{box}^+\,\mathsf{in}_d\, t]\!]_i$ is the d'th injection into the function with j'th component $[\![t]\!]_j$, and likewise for $[\![\mathsf{in}_d\,\mathsf{box}\, t]\!]_i$. □

2.3.3 Adequacy and Normalisation

We now define a logical relation between our denotational semantics and terms, from which both normalisation and adequacy will follow. Doing this inductively proves rather delicate, because induction on size will not support reasoning about our values, as fold refers to a larger type in its premise. This motivates a notion of *unguarded size* under which $A[\mu\alpha.A/\alpha]$ is 'smaller' than $\mu\alpha.A$. But under this metric $\blacktriangleright A$ is smaller than A, so next now poses a problem. But the meaning of $\blacktriangleright A$ at index $i+1$ is determined by A at index i, and so, as in Birkedal et al. [11], our relation will also induct on index. This in turn creates problems with box, whose meaning refers to all indexes

simultaneously, motivating a notion of *box depth*, allowing us finally to attain well-defined induction.

Definition 2.22. The *unguarded size* us of an open type follows the obvious definition for type size, except that $\mathsf{us}(\blacktriangleright A) = 0$.

The *box depth* bd of an open type is

- $\mathsf{bd}(A) = 0$ for $A \in \{\alpha, \mathbf{0}, \mathbf{1}, \mathbf{N}\}$;
- $\mathsf{bd}(A \times B) = \min(\mathsf{bd}(A), \mathsf{bd}(B))$, and similarly for $A + B, A \to B$;
- $\mathsf{bd}(\mu\alpha.A) = \mathsf{bd}(A)$, and similarly for $\mathsf{bd}(\blacktriangleright A)$;
- $\mathsf{bd}(\blacksquare A) = \mathsf{bd}(A) + 1$.

Lemma 2.23.

1. *α guarded in A implies* $\mathsf{us}(A[B/\alpha]) \leq \mathsf{us}(A)$.
2. $\mathsf{bd}(B) \leq \mathsf{bd}(A)$ *implies* $\mathsf{bd}(A[B/\alpha]) \leq \mathsf{bd}(A)$

Proof. By induction on the construction of the type A.

(i) follows with only interesting case the variable case – A cannot be α because of the requirement that α be guarded in A.

(ii) follows with interesting cases: variable case enforces $bd(B) = 0$; binary type-formers $\times, \to$ have for example $\mathsf{bd}(A_1) \geq \mathsf{bd}(A_1 \times A_2)$, so $\mathsf{bd}(A_1) \geq bd(B)$ and the induction follows; $\blacksquare A$ by construction has no free variables. □

Definition 2.24. The family of relations R_i^A, indexed by closed types A and positive integers i, relates elements of the semantics $a \in [\![A]\!]_i$ and closed typed terms $t : A$ and is defined as

- $nR_i^{\mathbf{N}}t$ iff $t \rightsquigarrow \mathsf{succ}^n \mathsf{zero}$;
- $*R_i^{\mathbf{1}}t$ iff $t \rightsquigarrow \langle\rangle$;
- $(a_1, a_2)R_i^{A_1 \times A_2}t$ iff $t \rightsquigarrow \langle t_1, t_2\rangle$ and $a_1 R_i^{A_1} t_1$ and $a_2 R_i^{A_2} t_2$;
- $[a, d]R_i^{A_1 + A_2}t$ iff $t \rightsquigarrow \mathsf{in}_d\, u$ for $d \in \{1, 2\}$, and $aR_i^{A_d}u$.
- $f R_i^{A \to B} t$ iff $t \rightsquigarrow \lambda x.s$ and for all $j \leq i$, $aR_j^A u$ implies $f_j(a)R_j^B s[u/x]$;
- $aR_i^{\mu\alpha.A}t$ iff $t \rightsquigarrow \mathsf{fold}\, u$ and $h_i(a)R_i^{A[\mu\alpha.A/\alpha]}u$, where h is the "unfold" isomorphism for the recursive type (ref. Lemma 2.17);
- $aR_i^{\blacktriangleright A}t$ iff $t \rightsquigarrow \mathsf{next}\, u$ and, where $i > 1$, $aR_{i-1}^A u$.
- $aR_i^{\blacksquare A}t$ iff $t \rightsquigarrow \mathsf{box}\, u$ and for all j, $a_j R_j^A u$;

Note that $R_i^{\mathbf{0}}$ is (necessarily) everywhere empty.

The above is well-defined by induction on the lexicographic ordering on box depth, then index, then unguarded size. First, the $\blacksquare$ case strictly decreases box depth, and no other case increases it (ref. Lemma 2.23.2 for μ-types). Second, the $\blacktriangleright$ case strictly decreases index, and no other case increases it (disregarding $\blacksquare$). Finally, all other cases strictly decrease unguarded size, as seen via Lemma 2.23.1 for μ-types.

Lemma 2.25. *If $t \rightsquigarrow u$ and $aR_i^A u$ then $aR_i^A t$.*

Proof. All cases follow similarly; consider $A_1 \times A_2$. $(a_1, a_2) R_i^{A_1 \times A_2} u$ implies $u \rightsquigarrow \langle t_1, t_2 \rangle$, where this value obeys some property. But then $t \rightsquigarrow \langle t_1, t_2 \rangle$ similarly. □

Lemma 2.26. *$aR_{i+1}^A t$ implies $r_i^{[\![A]\!]}(a) R_i^A t$.*

Proof. Cases $\mathbf{N}, \mathbf{1}, \mathbf{0}$ are trivial. Cases $\times$ and $+$ follow by induction because restrictions are defined pointwise. Case μ follows by induction and the naturality of the isomorphism h. Case $\blacksquare A$ follows because $r_i^{[\![\blacksquare A]\!]}(a) = a$.

For $A \to B$ take $j \leq i$ and $a' R_j^A u$. By the downwards closure in the definition of $R_{i+1}^{A \to B}$ we have $f_j(a') R_j^B s[u/x]$. But $f_j = (r_i^{[\![A \to B]\!]}(f))_j$.

With $\blacktriangleright A$, case $i = 1$ is trivial, so take $i = j + 1$. $aR_{j+2}^{\blacktriangleright A} t$ means $t \rightsquigarrow \mathsf{next}\, u$ and $aR_{j+1}^A u$, so by induction $r_j^{[\![A]\!]}(a) R_j^A u$, so $r_{j+1}^{[\![\blacktriangleright A]\!]}(a) R_j^A u$ as required. □

Lemma 2.27. *If $aR_i^A t$ and A is constant, then $aR_j^A t$ for all j.*

Proof. Easy induction on types, ignoring $\blacktriangleright A$ and treating $\blacksquare A$ as a base case. □

We may now turn to the proof of the Fundamental Lemma.

Lemma 2.28 (Fundamental Lemma)**.** *Take $\Gamma = (x_1 : A_1, \ldots, x_m : A_m)$, $\Gamma \vdash t : A$, and closed typed terms $t_k : A_k$ for $1 \leq k \leq m$. Then for all i, if $a_k R_i^{A_k} t_k$ for all k, then*

$$[\![\Gamma \vdash t : A]\!]_i\, (\vec{a})\, R_i^A\, t[\vec{t}/\vec{x}].$$

Proof. By induction on the typing $\Gamma \vdash t : A$. $\langle\rangle, \mathsf{zero}$ cases are trivial, and $\langle u_1, u_2 \rangle, \mathsf{in}_d\, t, \mathsf{fold}\, t$ cases follow by easy induction.

$\mathsf{succ}\, t$: If $t[\vec{t}/\vec{x}]$ reduces to $\mathsf{succ}^l\, \mathsf{zero}$ for some l then $\mathsf{succ}\, t[\vec{t}/\vec{x}]$ reduces to $\mathsf{succ}^{l+1}\, \mathsf{zero}$, as we may reduce under the succ.

$\pi_d t$ for $d \in \{1, 2\}$: If $[\![t]\!]_i\, (\vec{a}) R_i^{A_1 \times A_2} t[\vec{t}/\vec{x}]$ then $t[\vec{t}/\vec{x}] \rightsquigarrow \langle u_1, u_2 \rangle$ and u_d is related to the d'th projection of $[\![t]\!]_i\, (\vec{a})$. But then $\pi_d t[\vec{t}/\vec{x}] \rightsquigarrow \pi_d \langle u_1, u_2 \rangle \mapsto u_d$, so Lemma 2.25 completes the case.

abort: The induction hypothesis states that $[\![t]\!]_k\, (\vec{a}) R_k^{\mathbf{0}} t[\vec{t}/\vec{x}]$, but this is not possible, so the statement holds vacuously.

$\mathsf{case}\, t\, \mathsf{of}\, y_1.u_1; y_2.u_2$: If $[\![t]\!]_i\,(\vec{a}) R_i^{A_1+A_2} t[\vec{t}/\vec{x}]$ then $t[\vec{t}/\vec{x}] \rightsquigarrow \mathsf{in}_d\, u$ for some $d \in \{1,2\}$, with $[\![t]\!]_i\,(\vec{a}) = [a,d]$ and $a R_i^{A_d} u$. Then $[\![u_d]\!]_i\,(\vec{a},a) R_k^A u_d[\vec{t}/\vec{x}, u/y_d]$. Now we have that

$$(\mathsf{case}\, t\, \mathsf{of}\, y_1.u_1; y_2.u_2)[\vec{t}/\vec{x}] \rightsquigarrow \mathsf{case}\, \mathsf{in}_d\, u\, \mathsf{of}\, y_1.(u_1[\vec{t}/\vec{x}]); y_2.(u_2[\vec{t}/\vec{x}]),$$

which in turn reduces to $u_d[\vec{t}/\vec{x}, u/y_i]$, and Lemma 2.25 completes.

$\lambda x.t$: Taking $j \le i$ and $a R_j^A u$, we must show that $[\![\lambda x.t]\!]_i\,(\vec{a})_j(a) R_j^B t[\vec{t}/\vec{x}][u/x]$. The left hand side is $[\![t]\!]_j\,(\restriction_j(\vec{a}), a)$. For each k, $a_k \restriction R_j^{A_k} t_k$ by Lemma 2.26, and induction completes the case.

$u_1 u_2$: By induction $u_1[\vec{t}/\vec{x}] \rightsquigarrow \lambda x.s$ and $[\![u_1]\!]_k\,(\vec{a})_k([\![u_2]\!]_k\,(\vec{a})) R_i^B s[u_2[\vec{t}/\vec{x}]/x]$. Now we have $(u_1 u_2) \rightsquigarrow (\lambda x.s)(u_2[\vec{t}/\vec{x}]) \mapsto s[u_2[\vec{t}/\vec{x}]/x]$, and Lemma 2.25 completes.

$\mathsf{unfold}\, t$: we reduce under unfold, then reduce unfoldfold, then use Lemma 2.25.

$\mathsf{next}\, t$: Trivial for index 1. For $i = j+1$, if each $a_k R_{j+1}^{A_k} t_k$ then by Lemma 2.26 $r_j^{[\![A_k]\!]}(a_k) R_j^{A_k} t_k$. Then by induction $[\![t]\!]_j \circ r_j^{[\![\Gamma]\!]}(\vec{a}) R_j^A t[\vec{t}/\vec{x}]$, whose left side is by naturality $r_j^{[\![A]\!]} \circ [\![t]\!]_{j+1}\,(\vec{a}) = [\![\mathsf{next}\, t]\!]_{j+1}\,(\vec{a})$.

$\mathsf{prev}[\vec{y} \leftarrow \vec{u}].t$: $[\![u_k]\!]_i\,(\vec{a}) R_i^{A_k} u_k[\vec{t}/\vec{x}]$ by induction, so $[\![u_k]\!]_i\,(\vec{a}) R_{i+1}^{A_k} u_k[\vec{t}/\vec{x}]$ by Lemma 2.27. Then $[\![t]\!]_{i+1}\,([\![u_1]\!]_i\,(\vec{a}),\ldots) R_{i+1}^{\blacktriangleright A} t[u_1[\vec{t}/\vec{x}]/y_1,\ldots]$ by induction, so we have $t[u_1[\vec{t}/\vec{x}]/y_1,\ldots] \rightsquigarrow \mathsf{next}\, s$ with $[\![t]\!]_{i+1}\,([\![u_1]\!]_k\,(\vec{a}),\ldots) R_i^A s$. The left hand side is $[\![\mathsf{prev}[\vec{y} \leftarrow \vec{u}].t]\!]_i\,(\vec{a})$, while $\mathsf{prev}[\vec{y} \leftarrow \vec{u}[\vec{t}/\vec{x}]].t \mapsto \mathsf{prev}\, t[u_1[\vec{t}/\vec{x}]/y_1,\ldots] \rightsquigarrow \mathsf{prev}\, \mathsf{next}\, s \mapsto s$, so Lemma 2.25 completes.

$u_1 \circledast u_2$: Index 1 is trivial so set $i = j+1$. $[\![u_2]\!]_{j+1}\,(\vec{a}) R_{j+1}^{\blacktriangleright A} u_2[\vec{t}/\vec{x}]$ implies $u_2[\vec{t}/\vec{x}] \rightsquigarrow \mathsf{next}\, s_2$ with $[\![u_2]\!]_{j+1}\,(\vec{a}) R_j^A s_2$. Similarly $u_1 \rightsquigarrow \mathsf{next}\, s_1$ and $s_1 \rightsquigarrow \lambda x.s$ with $([\![u_1]\!]_{j+1}\,(\vec{a})_j) \circ [\![u_2]\!]_{j+1}\,(\vec{a}) R_j^B s[s_2/x]$. The left hand side is exactly $[\![u_1 \circledast u_2]\!]_{j+1}\,(\vec{a})$. Now $u_1 \circledast u_2 \rightsquigarrow \mathsf{next}\, s_1 \circledast u_2 \rightsquigarrow \mathsf{next}\, s_1 \circledast \mathsf{next}\, s_2 \mapsto \mathsf{next}(s_1 s_2)$, and $s_1 s_2 \rightsquigarrow (\lambda x.s) s_2 \mapsto s[s_2/x]$, completing the proof.

$\mathsf{box}[\vec{y} \leftarrow \vec{u}].t$: To show $[\![\mathsf{box}[\vec{y} \leftarrow \vec{u}].t]\!]_i\,(\vec{a}) R_i^{\blacksquare A} \mathsf{box}[\vec{y} \leftarrow \vec{u}].t)[\vec{t}/\vec{x}]$, we observe that the right hand side reduces in one step to $\mathsf{box}\, t[u_1[\vec{t}/\vec{x}]/y_1,\ldots]$. The j'th element of the left hand side is $[\![t]\!]_j\,([\![u_1]\!]_k\,(\vec{a}),\ldots)$. We need to show this is related by R_j^A to $t[u_1[\vec{t}/\vec{x}]/y_1,\ldots]$; this follows by Lemma 2.27 and induction.

$\mathsf{unbox}\, t$: By induction $t[\vec{t}/\vec{x}] \rightsquigarrow \mathsf{box}\, u$, so $\mathsf{unbox}\, t[\vec{t}/\vec{x}] \rightsquigarrow \mathsf{unbox}\, \mathsf{box}\, u \mapsto u$. By induction $[\![t]\!]_i\,(\vec{a})_i R_i^A u$, so $[\![\mathsf{unbox}\, t]\!]_i\,(\vec{a}) R_i^A u$, and Lemma 2.25 completes.

$\mathsf{box}^+[\vec{y} \leftarrow \vec{u}].t$: $[\![u_k]\!]_i\,(\vec{a}) R_i^{A_k} u_k[\vec{t}/\vec{x}]$ by induction, so $[\![u_k]\!]_i\,(\vec{a}) R_j^{A_k} u_k[\vec{t}/\vec{x}]$ for any j by Lemma 2.27. By induction $[\![t]\!]_j\,([\![u_1]\!]_k\,(\vec{a}),\ldots) R_j^{B_1+B_2} t[u_1[\vec{t}/\vec{x}]/y_1,\ldots]$. If $[\![t]\!]_j\,([\![u_1]\!]_k\,(\vec{a}),\ldots)$ is some $[b_j, d]$ we have $t[u_1[\vec{t}/\vec{x}]/y_1,\ldots] \rightsquigarrow \mathsf{in}_d\, s$ with $b_j R_j^{B_d} s$. Now $(\mathsf{box}^+[\vec{y} \leftarrow \vec{u}].t)[\vec{t}/\vec{x}] \mapsto \mathsf{box}^+\, t[u_1[\vec{t}/\vec{x}]/y_1,\ldots] \rightsquigarrow \mathsf{box}^+\, \mathsf{in}_d\, s$, which finally reduces to $\mathsf{in}_d\, \mathsf{box}\, s$, which yields the result. □

Theorem 2.29 (Adequacy and Normalisation)**.**

1. *For all closed terms* $\vdash t : A$ *it holds that* $[\![t]\!]_i \, R_i^A \, t$*;*
2. $[\![\vdash t : \mathbf{N}]\!]_i = n$ *implies* $t \rightsquigarrow \mathsf{succ}^n \, \mathsf{zero}$*;*
3. *All closed typed terms evaluate to a value.*

Proof. (1) specialises Lemma 2.28 to closed types. (2) and (3) hold by (1) and inspection of Definition 2.24. □

Definition 2.30. Typed *contexts* with typed holes are defined as obvious. Two terms $\Gamma \vdash t : A, \Gamma \vdash u : A$ are *contextually equivalent*, written $t \simeq_{\mathsf{ctx}} u$, if for all well-typed *closing* contexts C of type $\mathbf{N}$, the terms $C[t]$ and $C[u]$ reduce to the same value.

Corollary 2.31. $[\![t]\!] = [\![u]\!]$ *implies* $t \simeq_{\mathsf{ctx}} u$.

Proof. $[\![C[t]]\!] = [\![C[u]]\!]$ by compositionality of the denotational semantics. Then by Theorem 2.29.2 they reduce to the same value. □

2.4 Logic for the Guarded Lambda Calculus

In this section we will discuss the internal logic of the topos of trees, show that it yields a program logic $L\mathrm{g}\lambda$ which supports reasoning about the contextual equivalence of gλ-programs, remark on some properties of this program logic, and give some example proofs.

2.4.1 From Internal Logic to Program Logic

$\mathcal{S}$ is a presheaf category, and so a topos, and so its internal logic provides a model of higher-order logic with equality [61]. The internal logic of $\mathcal{S}$ has been explored elsewhere [14, 25, 58], but to motivate the results of this section we make some observations here.

As discussed in Example 2.16.2, the subobject classifier Ω is exactly the denotation of the *guarded conatural numbers* $\mathsf{CoNat}^{\mathrm{g}}$, as defined in the g$\lambda$-calculus in Section 2.2.4. The propositional connectives can then be defined via gλ-functions on the guarded conaturals: false $\bot$ is cozero, as defined in Section 2.2.4; true $\top$ is infinity; conjunction $\wedge$ is a minimum function readily definable on pairs of guarded conaturals; $\neg$ is

$$\lambda n.\, \mathsf{case}(\mathsf{unfold}\, n)\, \mathsf{of}\, x_1.\mathsf{infinity};\, x_2.\mathsf{cozero} \;:\; \mathsf{CoNat}^{\mathrm{g}} \to \mathsf{CoNat}^{\mathrm{g}}$$

and so on. The connectives $\forall x : A$, $\exists x : A$, and $=_A$ cannot be expressed as gλ-functions for an arbitrary gλ-type A, but are definable as (parametrised) operations on Ω in the usual way [61, Section IV.9].

Along with the standard connectives we can define a *modality* $\triangleright$, whose action on the subobject classifier corresponds precisely to the function cosucc on guarded conaturals defined in Section 2.2.4. We call this modality 'later', overloading our name for our type-former $\blacktriangleright$, and the functor on $\mathcal{S}$ with the same name and symbol introduced in Definition 2.14.3. This overloading is justified by a tight relationship between these concepts which we will investigate below. For now, note that cosucc can be defined as a composition of functions $\mathsf{lift} \circ \mathsf{next}$, where lift[5] is a function $\blacktriangleright\Omega \to \Omega$ definable in the gλ calculus as

$$\lambda n.\, \mathsf{fold}(\mathsf{in}_2\, n) \;:\; \blacktriangleright \mathsf{CoNat}^{\mathrm{g}} \to \mathsf{CoNat}^{\mathrm{g}}$$

Further, $\mathsf{infinity}$ is $\mathsf{fix}\,\mathsf{lift}$. We will make use of this lift function later in this section.

Returning to the propositional connectives, double negation $\neg\neg$ corresponds to the gλ-function

$$\lambda n.\, \mathsf{case}(\mathsf{unfold}\, n)\, \mathsf{of}\, x_1.\mathsf{cozero};\, x_2.\mathsf{infinity} \;:\; \mathsf{CoNat}^{\mathrm{g}} \to \mathsf{CoNat}^{\mathrm{g}}$$

Now consider the poset $\mathsf{Sub}(X)$ of subobjects of X, which are pointwise subsets whose restriction maps are determined by the restriction maps of X; or equivalently, characteristic arrows $X \to \Omega$. The function $\neg\neg : \Omega \to \Omega$ extends to a monotone function $\mathsf{Sub}(X) \to \mathsf{Sub}(X)$ by composition with characteristic arrows as obvious. This function preserves joins, and so by the adjoint functor theorem for posets has a right adjoint $\mathsf{Sub}(X) \to \mathsf{Sub}(X)$, which we write $\Box$ and call 'always' [20]. The notational similarity with the type-former and functor $\blacksquare$ is, as with $\triangleright$ and $\blacktriangleright$, deliberate and will be explored further. First, we can offer a more concrete definition of $\Box$:

Definition 2.32.

- Take a $\mathcal{S}$-object X, positive integer m, and element $x \in X_m$, and recall that for any $n \geq m$ the function $\restriction_m : X_n \to X_m$ is defined by composing restriction functions. Then the *height* of x in X, written $\mathsf{height}_X(x)$, is the largest integer $n \geq m$ such that there exists $y \in X_n$ with $\restriction_m(y) = x$, or ∞ if there is no such largest n.

- Given a subobject Y of X, the characteristic arrow of the subobject $\Box Y$ of X is defined as

$$(\chi_{\Box Y})_n(x) \;=\; \begin{cases} (\chi_Y)_n(x) & \mathsf{height}_Y(x) = \mathsf{height}_X(x) \\ 0 & \text{otherwise.} \end{cases}$$

The condition regarding the height of elements allows the modality $\Box$ to reflect the global, rather than pointwise, structure of a subobject. For example, considering the object $\blacktriangleright 0$, which is a singleton at its first stage and empty set at all later stages, as a subobject of the terminal object 1, the subobject $\Box(\blacktriangleright 0)$ is 0.

[5] called succ by Birkedal et al. [14]; we avoid this because of the clash with the name for a term-former.

Example 2.33. A proposition φ with no free variables corresponds in the internal logic of $\mathcal{S}$ to an arrow $1 \to \Omega$, which as we have seen in turn corresponds to a guarded conatural number. The proposition $\Box\varphi$ also corresponds a guarded conatural number, so we can see the action of $\Box$ *on closed propositions* as arising from a function $\mathbb{N} + \{\infty\} \to \mathbb{N} + \{\infty\}$ defined by

$$\Box(n) = \begin{cases} \infty & n = \infty \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

This is a perfectly good function in **Set**, but it does *not* correspond to an $\mathcal{S}$-arrow $\Omega \to \Omega$, because it is hopelessly unproductive – we need to make infinitely many observations of the input before we decide anything about the output. Similarly, we cannot define a function of the type $\mathsf{CoNat}^{\mathsf{g}} \to \mathsf{CoNat}^{\mathsf{g}}$ in the gλ-calculus with this behaviour.

The case where we have a subobject Y of a *constant* object X is similar to the case of subobjects of 1 – the characteristic function of $\Box Y$ maps each element x of X to a conatural number, which is then composed with the $\Box$ function (2.2).

Note further than $\Box$ does not commute with substitution; in particular, given a substitution σ, $\Box(\varphi\sigma)$ does not necessarily imply $(\Box\varphi)\sigma$. However these formulae *are* equivalent if σ is a substitution between constant contexts. In practice we will use $\Box$ only in constant context.

We may now proceed to the definition of the program logic $L\mathrm{g}\lambda$:

Definition 2.34. $L\mathrm{g}\lambda$ is the typed higher order logic with equality defined by the internal logic of $\mathcal{S}$, whose types and function symbols are the types and term-formers of the gλ-calculus, interpreted in $\mathcal{S}$ as in Section 2.3.2, and further extended by the modalities $\triangleright, \Box$.

We write $\Gamma \mid \Xi \vdash \varphi$ where the proposition φ with term variables drawn from the context Γ is entailed by the set of propositions Ξ. Note that we use the symbol Ω for the type of propositions, although this is precisely the denotation of the guarded conatural numbers.

This logic may be used to prove contextual equivalence of programs:

Theorem 2.35. *Let t_1 and t_2 be two* gλ *terms of type A in context Γ. If the sequent $\Gamma \mid \emptyset \vdash t_1 =_A t_2$ is provable, then t_1 and t_2 are contextually equivalent.*

Proof. Recall that equality in the internal logic of a topos is just equality of morphisms. Hence t_1 and t_2 denote same morphism from $[\![\Gamma]\!]$ to $[\![A]\!]$. Adequacy (Corollary 2.31) then implies that t_1 and t_2 are contextually equivalent. □

2.4.2 Properties of the Logic

The definition of the logic $L\mathrm{g}\lambda$ from the previous section establishes its syntax, and semantics in the topos of trees, without giving much sense of how proofs might be constructed. Clouston and Goré [25] have provided a sound and complete sequent calculus, and hence decision procedure, for the fragment of the internal logic of $\mathcal{S}$ with propositional connectives and $\triangleright$, but the full logic $L\mathrm{g}\lambda$ is considerably more expressive than this; for example it is not decidable [67]. In this section we will establish some reasoning principles for $L\mathrm{g}\lambda$, which will assist us in the next section in constructing proofs about $\mathrm{g}\lambda$-programs.

We start by noting that the usual $\beta\eta$-laws and commuting conversions for the λ-calculus with products, sums, and iso-recursive types hold. These may be extended with new equations for the new $\mathrm{g}\lambda$-constructs, sound in the model $\mathcal{S}$, as listed in Figure 2.3.

Many of the rules of Figure 2.3 are unsurprising, adding η-rules to the β-rules of Definition 2.2, noting only that in the case of $\blacktriangleright$ we use the rule of equation (2.1), because we are here allowing the consideration of open terms. The reduction rule for $\circledast$ is joined by the 'composition' equality for applicative functors [64]. In addition to the β-rule for box^+ of Definition 2.12, which govern how this connective commutes with the constructors in_1, in_2 and box, we also add a rule showing how it interacts with the eliminators case and unbox. The next rule resembles a traditional commuting conversion for case with box^+, but specialised to hold where the sum $C + D$ on which the case split occurs has constant type.

There are finally three rules showing how substitutions can be moved in and out of the explicit substitutions attached to the term-formers prev, box, and box^+, provided everything is suitably constant. Because of these operators' binding structure, substituted terms can get 'stuck' inside explicit substitutions and so cannot interact with the terms the operators are applied to. This is essential for soundness in general, but not where everything is suitably constant, in which case these rules become essential to further simplifying terms. As an example, the rather complicated commuting conversion for Intuitionistic S4 defined by Bierman and de Paiva [9]

$$\mathsf{box}[\vec{x} \leftarrow \vec{t}, \vec{y} \leftarrow \vec{u}].(t[\mathsf{box}\, \iota.u/x]) \quad \approx \quad \mathsf{box}[\vec{x} \leftarrow \vec{t}, x \leftarrow (\mathsf{box}[\vec{y} \leftarrow \vec{u}].u)].t$$

comes as a corollary.

We now pick out a distinguished class of $\mathcal{S}$-objects and $\mathrm{g}\lambda$-types that enjoy extra properties that are useful in some $L\mathrm{g}\lambda$ proofs.

Definition 2.36. An $\mathcal{S}$-object is *total and inhabited* if all its restriction functions are surjective, and all its sets are non-empty.

A $\mathrm{g}\lambda$-type is *total and inhabited* if its denotation in $\mathcal{S}$ is total and inhabited.

In fact we can express this property directly in the internal logic:

$$\frac{\vec{x}:\vec{A}\vdash t:A \qquad \Gamma\vdash\vec{t}:\vec{A}}{\Gamma\vdash \mathsf{prev}[x_1\leftarrow t_1,\dots,x_n\leftarrow t_n].(\mathsf{next}\,t)=t\left[\vec{t}/\vec{x}\right]}$$

$$\frac{\vec{x}:\vec{A}\vdash t:\blacktriangleright A \qquad \Gamma\vdash\vec{t}:\vec{A}}{\Gamma\vdash \mathsf{next}(\mathsf{prev}[x_1\leftarrow t_1,\dots,x_n\leftarrow t_n].t)=t\left[\vec{t}/\vec{x}\right]} \qquad \frac{\Gamma\vdash t_1:A\to B \qquad \Gamma\vdash t_2:A}{\Gamma\vdash \mathsf{next}\,t_1\circledast\mathsf{next}\,t_2=\mathsf{next}(t_1\,t_2)}$$

$$\frac{\Gamma\vdash f:\blacktriangleright(B\to C) \qquad \Gamma\vdash g:\blacktriangleright(A\to B) \qquad \Gamma\vdash t:\blacktriangleright A}{\Gamma\vdash f\circledast(g\circledast t)=(\mathsf{next}\,\mathsf{comp})\circledast f\circledast g\circledast t}$$

$$\frac{\vec{x}:\vec{A}\vdash t:A \qquad \Gamma\vdash\vec{t}:\vec{A}}{\Gamma\vdash \mathsf{unbox}(\mathsf{box}[\vec{x}\leftarrow\vec{t}].t)=t\left[\vec{t}/\vec{x}\right]} \qquad \frac{\vec{x}:\vec{A}\vdash t:\blacksquare A \qquad \Gamma\vdash\vec{t}:\vec{A}}{\Gamma\vdash \mathsf{box}[\vec{x}\leftarrow\vec{t}].\,\mathsf{unbox}\,t=t\left[\vec{t}/\vec{x}\right]}$$

$$\frac{\vec{x}:\vec{A}\vdash t:A \qquad \Gamma\vdash\vec{t}:\vec{A}}{\Gamma\vdash \mathsf{box}^+[\vec{x}\leftarrow\vec{t}].\,\mathsf{in}_1\,t=\mathsf{in}_1\,\mathsf{box}[\vec{x}\leftarrow\vec{t}].t} \qquad \frac{\vec{x}:\vec{A}\vdash t:B \qquad \Gamma\vdash\vec{t}:\vec{A}}{\Gamma\vdash \mathsf{box}^+[\vec{x}\leftarrow\vec{t}].\,\mathsf{in}_2\,t=\mathsf{in}_2\,\mathsf{box}[\vec{x}\leftarrow\vec{t}].t}$$

$$\frac{\vec{x}:\vec{A}\vdash t:A+B \qquad \Gamma\vdash\vec{t}:\vec{A} \qquad \Gamma,z_1:A\vdash u_1:C \qquad \Gamma,z_2:B\vdash u_2:C}{\begin{array}{l}\Gamma\vdash \mathsf{case}(\mathsf{box}^+[\vec{x}\leftarrow\vec{t}].t)\,\mathsf{of}\,y_1.u_1[\mathsf{unbox}\,y_1/z_1];y_2.u_2[\mathsf{unbox}\,y_2/z_2]\\ \quad=\mathsf{case}(t[\vec{t}/\vec{x}])\,\mathsf{of}\,z_1.u_1;z_2.u_2\end{array}}$$

$$\frac{\vec{x}:\vec{A}\vdash t:C+D \qquad \Gamma\vdash\vec{t}:\vec{A} \qquad \vec{x}:\vec{A},y_1:C\vdash u_1:A+B \qquad \vec{x}:\vec{A},y_2:D\vdash u_2:A+B}{\begin{array}{l}\Gamma\vdash \mathsf{box}^+[\vec{x}\leftarrow\vec{t}].\,\mathsf{case}\,t\,\mathsf{of}\,y_1.u_1;y_2.u_2\\ \quad=\mathsf{case}(t[\vec{t}/\vec{x}])\,\mathsf{of}\,y_1.\,\mathsf{box}^+[\vec{x},y_1\leftarrow\vec{t},y_1].u_1;y_2.\,\mathsf{box}^+[\vec{x},y_2\leftarrow\vec{t},y_2].u_2\end{array}}$$

$$\frac{\vec{x}:\vec{A}\vdash t:\blacktriangleright A \qquad \vec{y}:\vec{B}\vdash\vec{t}:\vec{A} \qquad \Gamma\vdash\vec{u}:\vec{B}}{\Gamma\vdash \mathsf{prev}[\vec{y}\leftarrow\vec{u}].(t[\vec{t}/\vec{x}])=\mathsf{prev}[\vec{x}\leftarrow(\vec{t}[\vec{u}/\vec{x}])].t}$$

$$\frac{\vec{x}:\vec{A}\vdash t:A \qquad \vec{y}:\vec{B}\vdash\vec{t}:\vec{A} \qquad \Gamma\vdash\vec{u}:\vec{B}}{\Gamma\vdash \mathsf{box}[\vec{y}\leftarrow\vec{u}].(t[\vec{t}/\vec{x}])=\mathsf{box}[\vec{x}\leftarrow(\vec{t}[\vec{u}/\vec{x}])].t}$$

$$\frac{\vec{x}:\vec{A}\vdash t:A+B \qquad \vec{y}:\vec{B}\vdash\vec{t}:\vec{A} \qquad \Gamma\vdash\vec{u}:\vec{B}}{\Gamma\vdash \mathsf{box}^+[\vec{y}\leftarrow\vec{u}].(t[\vec{t}/\vec{x}])=\mathsf{box}^+[\vec{x}\leftarrow(\vec{t}[\vec{u}/\vec{x}])].t}$$

Figure 2.3: Equations between gλ-terms in $Lg\lambda$. Types in $\vec{A},\vec{B},C,D$ are assumed constant. comp is the composition $\lambda x.\lambda y.\lambda z.x(yz):(B\to C)\to(A\to B)\to(A\to C)$.

Lemma 2.37. *A type A is total and inhabited iff the formula*

$$\mathrm{TI}(A) \triangleq \forall a' : \blacktriangleright A, \exists a : A, a' =_{\blacktriangleright A} \mathsf{next}\, a$$

is valid.

Proof. The formula $\mathrm{TI}(A)$ expresses the *internal surjectivity* of the $\mathcal{S}$-arrow $\mathsf{next} : [\![A]\!] \to \blacktriangleright [\![A]\!]$. In any presheaf topos, this holds of an arrow precisely when its components are all surjective. It hence suffices to show that any $\mathcal{S}$-object X is total and inhabited iff all the functions of $\mathsf{next} : X \to \blacktriangleright X$ are surjective: X_1 is non-empty iff $! : X_1 \to (\blacktriangleright X)_1 = \{*\}$ is surjective, all other arrows of next are the restriction functions themselves, and if X_1 is non-empty and all restriction functions are surjective, then all X_i are non-empty. □

In fact almost all gλ-types are total and inhabited, as the next lemma and its corollary show:

Lemma 2.38. *Let $F : (\mathcal{S}^{op} \times \mathcal{S})^{n+1} \to \mathcal{S}$ be a* locally contractive *[14, Definition II.10] functor that maps tuples of total and inhabited objects to total and inhabited objects, i.e. F restricts to the full subcategory $ti\mathcal{S}$ of total and inhabited $\mathcal{S}$-objects.*

Then its fixed point $\mathsf{Fix}(F) : (\mathcal{S}^{op} \times \mathcal{S})^{n} \to \mathcal{S}$ is also total and inhabited.

Proof. $ti\mathcal{S}$ is equivalent to the category of bisected complete non-empty ultrametric spaces $\mathcal{M}$ [14, Section 5]. $\mathcal{M}$ is known to be an M-category in the sense of Birkedal et al. [12] and it is easy to see that locally contractive functors in $\mathcal{S}$ are locally contractive in the M-category sense. Because fixed points exist in M-categories, the fixed point of F exists in $ti\mathcal{S}$. □

Corollary 2.39. *All gλ-types that do not have the empty type* **0** *in their syntax tree are total and inhabited.*

Proof. The μ-case is covered by Lemma 2.38, because open types whose free variables are guarded denote locally contractive functors; the $\blacksquare$ case holds because total and inhabited objects X admit at least one global element $1 \to X$; all other cases are routine. □

Further sound reasoning principles in $L\mathrm{g}\lambda$, some making use of the concept of total and inhabited type, are listed in Figure 2.4, and in the lemmas below, whose proofs are all routine. Note that the rule $\mathrm{EQ}^{\triangleright}_{\mathsf{next}}$ establishes a close link between $\blacktriangleright$ and $\triangleright$, as Lemma 2.41 does for $\blacksquare$ and $\square$.

Lemma 2.40. *For any type A and term $f : \blacktriangleright A \to A$ we have $\mathsf{fix}\, f =_A f\,(\mathsf{next}(\mathsf{fix}\, f))$ and, if u is any other term such that $f(\mathsf{next}\, u) =_A u$, then $u =_A \mathsf{fix}\, f$.* □

Finally, in the next section we will come to the problem of proving $x =_{\blacksquare A} y$ from $\mathsf{unbox}\, x =_A \mathsf{unbox}\, y$. This does not hold in general, but using the semantics of $L\mathrm{g}\lambda$ we can prove the proposition below.

Lemma 2.41. *The formula $\square(\mathsf{unbox}\, x =_A \mathsf{unbox}\, y) \Rightarrow x =_{\blacksquare A} y$ is valid.* □

$$\frac{}{\Gamma \mid \Xi, (\triangleright\varphi \Rightarrow \varphi) \vdash \varphi}\ \text{Löb} \qquad \frac{}{\Gamma, x : X \mid \exists y : Y, \triangleright\varphi(x,y) \vdash \triangleright(\exists y : Y, \varphi(x,y))}\ \exists\triangleright$$

$$\frac{}{\Gamma, x : X \mid \triangleright(\forall y : Y, \varphi(x,y)) \vdash \forall y : Y, \triangleright\varphi(x,y)}\ \forall\triangleright \qquad \frac{}{\Gamma \mid \Xi, \varphi \vdash \triangleright\varphi}$$

$$\frac{\star \in \{\wedge, \vee, \Rightarrow\}}{\Gamma \mid \triangleright(\varphi \star \psi) \dashv\vdash \triangleright\varphi \star \triangleright\psi} \qquad \frac{\Gamma \mid \neg\neg\varphi \vdash \psi}{\Gamma \mid \varphi \vdash \Box\psi} \qquad \frac{\Gamma \mid \varphi \vdash \Box\psi}{\Gamma \mid \neg\neg\varphi \vdash \psi} \qquad \frac{\Gamma \mid \varphi \vdash \psi}{\Gamma \mid \Box\varphi \vdash \Box\psi}$$

$$\frac{}{\Gamma \mid \Box\varphi \vdash \varphi} \qquad \frac{}{\Gamma \mid \Box\varphi \vdash \Box\Box\varphi}$$

$$\frac{}{\forall x, y : X. \triangleright(x =_X y) \Leftrightarrow \mathsf{next}\, x =_{\blacktriangleright X} \mathsf{next}\, y}\ \mathrm{EQ}^{\triangleright}_{\mathsf{next}}$$

Figure 2.4: Valid rules for $\triangleright$ and $\Box$. The converse entailment in $\forall\triangleright$ and $\exists\triangleright$ rules holds if Y is total and inhabited. In all rules involving $\Box$ the context Γ is assumed constant.

2.4.3 Examples

In this section we see examples of $L\mathrm{g}\lambda$ proofs regarding gλ-programs.

Example 2.42.

1. For any $f : A \to B$ and $g : B \to C$ we have

$$(\mathsf{map}^{\mathrm{g}}\, f) \circ (\mathsf{map}^{\mathrm{g}}\, g) =_{\mathsf{Str}^{\mathrm{g}} A \to \mathsf{Str}^{\mathrm{g}} C} \mathsf{map}^{\mathrm{g}}(f \circ g). \tag{2.3}$$

   Equality of functions is extensional, so it suffices to show that these are equal on any stream of type $\mathsf{Str}^{\mathrm{g}} A$, for which we use the variable s. The proof proceeds by unfolding the definitions on each side, observing that the heads are equal, then proving equality of the tails by *Löb induction*; i.e. our induction hypothesis will be (2.3) with $\triangleright$ in front:

$$\triangleright((\mathsf{map}^{\mathrm{g}}\, f) \circ (\mathsf{map}^{\mathrm{g}}\, g) = \mathsf{map}^{\mathrm{g}}(f \circ g)). \tag{2.4}$$

   Now unfolding the left hand side of (2.3) applied to s, using the definition of $\mathsf{map}^{\mathrm{g}}$ from Example 2.9.5, along with β-rules and Lemma 2.40, we get

$$f(g(\mathsf{hd}^{\mathrm{g}}\, s)) :: (\mathsf{next}(\mathsf{map}^{\mathrm{g}}\, f) \circledast ((\mathsf{next}(\mathsf{map}^{\mathrm{g}}\, g)) \circledast \mathsf{tl}^{\mathrm{g}}\, s))$$

   By applying the composition rule for $\circledast$ this simplifies to

$$f(g(\mathsf{hd}^{\mathrm{g}}\, s)) :: ((\mathsf{next}\, \mathsf{comp}) \circledast (\mathsf{next}(\mathsf{map}^{\mathrm{g}}\, f)) \circledast (\mathsf{next}(\mathsf{map}^{\mathrm{g}}\, g)) \circledast \mathsf{tl}^{\mathrm{g}}\, s)$$

Applying the reduction rule for $\circledast$ we simplify this further to

$$f(g(\mathsf{hd}^{\mathsf{g}}\, s)) :: (\mathsf{next}((\mathsf{map}^{\mathsf{g}}\, f) \circ (\mathsf{map}^{\mathsf{g}}\, g)) \circledast \mathsf{tl}^{\mathsf{g}}\, s) \tag{2.5}$$

Unfolding the right of (2.3) similarly, we get

$$f(g(\mathsf{hd}^{\mathsf{g}}\, s)) :: (\mathsf{next}(\mathsf{map}^{\mathsf{g}}(f \circ g)) \circledast \mathsf{tl}^{\mathsf{g}}\, s) \tag{2.6}$$

These streams have the same head; we proceed on the tail using our induction hypothesis (2.4). By $\textsc{eq}^{\triangleright}_{\mathsf{next}}$ we immediately have

$$\mathsf{next}((\mathsf{map}^{\mathsf{g}}\, f) \circ (\mathsf{map}^{\mathsf{g}}\, g)) = \mathsf{next}\,\mathsf{map}^{\mathsf{g}}(f \circ g)$$

replacing equals by equals then makes (2.5) equal to (2.6); Löb completes the proof.

2. We now show how $L_{\mathsf{g}}\lambda$ can prove a second-order property. Given a predicate P on a type A, that is, $P : A \to \Omega$, we can lift this to a predicate $P_{\mathsf{Str}^{\mathsf{g}}}$ on $\mathsf{Str}^{\mathsf{g}} A$ expressing that P holds for all elements of the stream by the definition

$$P_{\mathsf{Str}^{\mathsf{g}}} \triangleq \mathsf{fix}\, \lambda r.\lambda s.P(\mathsf{hd}^{\mathsf{g}}\, s) \wedge \mathsf{lift}\,(r \circledast (\mathsf{tl}^{\mathsf{g}}\, s)) \;:\; \mathsf{Str}^{\mathsf{g}}\mathbb{N} \to \Omega$$

We can now prove for a *total and inhabited* type A that

$$\begin{aligned}&\forall P, Q : (A \to \mathsf{CoNat}^{\mathsf{g}}), \forall f : A \to A, (\forall x : A, P(x) \Rightarrow Q(f(x)))\\ &\Rightarrow \forall s : \mathsf{Str}A, P_{\mathsf{Str}^{\mathsf{g}}}(s) \Rightarrow Q_{\mathsf{Str}^{\mathsf{g}}}(\mathsf{map}^{\mathsf{g}}\, f\, s).\end{aligned}$$

Recall that $\mathsf{map}^{\mathsf{g}}$ satisfies $\mathsf{map}^{\mathsf{g}}\, f\, s = f(\mathsf{hd}^{\mathsf{g}}\, s) :: (\mathsf{next}(\mathsf{map}^{\mathsf{g}}\, f) \circledast (\mathsf{tl}^{\mathsf{g}}\, s))$. We will prove the property by Löb induction, and so assume

$$\triangleright(\forall s : \mathsf{Str}\mathbb{N}, P_{\mathsf{Str}^{\mathsf{g}}}(s) \Rightarrow Q_{\mathsf{Str}^{\mathsf{g}}}(\mathsf{map}^{\mathsf{g}}\, f\, s)) \tag{2.7}$$

Let s be a stream satisfying $P_{\mathsf{Str}^{\mathsf{g}}}$. If we unfold $P_{\mathsf{Str}^{\mathsf{g}}}(s)$ we get $P(\mathsf{hd}^{\mathsf{g}}\, s)$ and $\mathsf{lift}(\mathsf{next}\, P_{\mathsf{Str}^{\mathsf{g}}} \circledast (\mathsf{tl}^{\mathsf{g}}\, s))$. We need to prove $Q(\mathsf{hd}^{\mathsf{g}}(\mathsf{map}^{\mathsf{g}}\, f\, s))$ and $\mathsf{lift}(\mathsf{next}\, Q_{\mathsf{Str}^{\mathsf{g}}} \circledast (\mathsf{tl}^{\mathsf{g}}(\mathsf{map}^{\mathsf{g}}\, f\, s)))$. The first is easy since $Q(\mathsf{hd}^{\mathsf{g}}(\mathsf{map}^{\mathsf{g}}\, f\, s)) = Q(f(\mathsf{hd}^{\mathsf{g}}\, s))$. For the second we have $\mathsf{tl}^{\mathsf{g}}(\mathsf{map}^{\mathsf{g}}\, f\, s) = \mathsf{next}(\mathsf{map}^{\mathsf{g}}\, f) \circledast (\mathsf{tl}^{\mathsf{g}}\, s)$. As A is total and inhabited, $\mathsf{Str}^{\mathsf{g}} A$ is also by Corollary 2.39. Hence there is a stream s' such that $\mathsf{next}\, s' = \mathsf{tl}^{\mathsf{g}}\, s$. This gives $\mathsf{tl}^{\mathsf{g}}(\mathsf{map}^{\mathsf{g}}\, f\, s) = \mathsf{next}(\mathsf{map}^{\mathsf{g}}\, f\, s')$ and so our desired result reduces to $\mathsf{lift}(\mathsf{next}(Q_{\mathsf{Str}^{\mathsf{g}}}(\mathsf{map}^{\mathsf{g}}\, f\, s')))$ and $\mathsf{lift}(\mathsf{next}\, P_{\mathsf{Str}^{\mathsf{g}}} \circledast (\mathsf{tl}^{\mathsf{g}}\, s))$ is equivalent to $\mathsf{lift}(\mathsf{next}(P_{\mathsf{Str}^{\mathsf{g}}}(s')))$. But $\mathsf{lift} \circ \mathsf{next} = \triangleright$ and so the induction hypothesis (2.7) and Löb finish the proof.

We now turn to examples that involve the constant type-former $\blacksquare$.

Example 2.43.

1. Recall the functions $\mathsf{iterate}' : (A \to A) \to A \to \mathsf{Str}^{\mathsf{g}} A$ of Example 2.9.6 and $\mathsf{every2nd} : \mathsf{Str} A \to \mathsf{Str}^{\mathsf{g}} A$ of Example 2.10.3. Then for every $x : A$ and $f : A \to A$,

$$\mathsf{every2nd}(\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x) =_{\mathsf{Str}^{\mathsf{g}} A} \mathsf{iterate}'\,f^2\,x$$

where f^2 is $\lambda x.f(f\,x)$.

First we prove the intermediate result

$$\mathsf{tl}(\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x) =_{\mathsf{Str} A} \mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,(f\,x) \tag{2.8}$$

which follows by:

$$\begin{aligned}
\mathsf{tl}\,(\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x) &= \mathsf{box}\,[s \leftarrow \mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x].\,\mathsf{prev}\,\iota.\,\mathsf{tl}^{\mathsf{g}}\,\mathsf{unbox}\,s \\
&= \mathsf{box}\,\iota.\,\mathsf{prev}[s \leftarrow \mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x].\,\mathsf{tl}^{\mathsf{g}}\,\mathsf{unbox}\,s \\
&= \mathsf{box}\,\iota.\,\mathsf{prev}\,\iota.\,\mathsf{tl}^{\mathsf{g}}\,\mathsf{unbox}\,\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x \\
&= \mathsf{box}\,\iota.\,\mathsf{prev}\,\iota.\,\mathsf{tl}^{\mathsf{g}}\,\mathsf{iterate}'\,f\,x \\
&= \mathsf{box}\,\iota.\,\mathsf{prev}\,\iota.(\mathsf{next}\,\mathsf{iterate}'\,f) \circledast (\mathsf{next}(f\,x)) \\
&= \mathsf{box}\,\iota.\,\mathsf{prev}\,\iota.\,\mathsf{next}(\mathsf{iterate}'\,f\,(f\,x)) \\
&= \mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,(f\,x)
\end{aligned}$$

The first step follows by the definition of tl and the β-rule for functions. The next two steps require the ability to move substitutions through a box and prev; see the last three equations of Figure 2.3. The remaining steps follow from unfolding definitions, various β-rules, and Lemma 2.40.

Now for Löb induction assume

$$\triangleright\Big(\mathsf{every2nd}(\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x) =_{\mathsf{Str}^{\mathsf{g}} A} \mathsf{iterate}'\,f^2\,x\Big), \tag{2.9}$$

then we can derive

$$\begin{aligned}
&\mathsf{every2nd}\,(\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x) \\
&\qquad = x :: (\mathsf{next}\,\mathsf{every2nd}) \circledast (\mathsf{next}\,\mathsf{tl}\,\mathsf{tl}\,\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x) \\
&\qquad = x :: \mathsf{next}\,\mathsf{every2nd}\,\mathsf{tl}\,\mathsf{tl}\,\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,x \\
&\qquad = x :: \mathsf{next}\,\mathsf{every2nd}\,\mathsf{box}\,\iota.\,\mathsf{iterate}'\,f\,(f^2\,x) && (2.8) \\
&\qquad = x :: \mathsf{next}\,\mathsf{iterate}'\,f^2\,(f^2\,x) && (2.9)\text{ and }\textsc{eq}^{\triangleright}_{\mathsf{next}} \\
&\qquad = \mathsf{iterate}'\,f^2\,x
\end{aligned}$$

One might wonder why we use iterate′ here instead of the more general iterate; the answer is that we cannot form the subterm $\mathsf{box}\,\iota.\,\mathsf{iterate}\,f\,x$ if f is a variable of type $\blacktriangleright(A \to A)$, because this is not a constant type.

2. Given a term in constant context $f : A \to B$ we define

$$\mathcal{L}(f) \triangleq \mathsf{lim}\,\mathsf{box}\,\iota.f : \blacksquare A \to \blacksquare B$$

recalling lim from Example 2.10.2. For any such f and $x : \blacksquare A$ we can then prove $\mathsf{unbox}(\mathcal{L}(f)\,x) =_B f(\mathsf{unbox}\,x)$. This allows us to prove, for example,

$$\mathcal{L}(f \circ g) = \mathcal{L}(f) \circ \mathcal{L}(g) \tag{2.10}$$

as follows: $\mathsf{unbox}(\mathcal{L}(f \circ g)(x)) = f \circ g(\mathsf{unbox}\,x) = \mathsf{unbox}(\mathcal{L}(f) \circ \mathcal{L}(g)(x))$. This is true without any assumptions, and so $\Box(\mathsf{unbox}(\mathcal{L}(f \circ g)(x)) = \mathsf{unbox}(\mathcal{L}(f) \circ \mathcal{L}(g)(x)))$, so by Lemma 2.41 and functional extensionality, (2.10) follows.

For functions of arity k we define $\mathcal{L}_k$ using $\mathcal{L}$, and analogous properties hold, e.g. we have $\mathsf{unbox}(\mathcal{L}_2(f)\,x\,y) = f(\mathsf{unbox}\,x)(\mathsf{unbox}\,y)$, which allows us to lift equalities proved for functions on guarded types to functions on constant types; see Section 2.5 for an example.

3. In Section 2.2.4 we claimed there is an isomorphism between the types $\blacksquare A + \blacksquare B$ and $\blacksquare(A + B)$, witnessed by the terms

$$\begin{aligned}&\lambda x.\,\mathsf{box}\,\iota.\,\mathsf{case}\,x\,\mathsf{of}\,x_1.\,\mathsf{in}_1\,\mathsf{unbox}\,x_1\,;x_2.\,\mathsf{in}_2\,\mathsf{unbox}\,x_2\\ &\quad : (\blacksquare A + \blacksquare B) \to \blacksquare(A + B)\end{aligned}$$

$$\begin{aligned}&\lambda x.\,\mathsf{box}^+\,\iota.\,\mathsf{unbox}\,x\\ &\quad : \blacksquare(A + B) \to \blacksquare A + \blacksquare B.\end{aligned}$$

We are now in a position to prove that these terms are mutually inverse. In the below we use the rules regarding the permutation of substitutions through box^+, the interaction of box^+ with case, and η-rules for sums and $\blacksquare$:

$$\begin{aligned}&(\lambda x.\,\mathsf{box}\,\iota.\,\mathsf{case}\,x\,\mathsf{of}\,x_1.\,\mathsf{in}_1\,\mathsf{unbox}\,x_1\,;x_2.\,\mathsf{in}_2\,\mathsf{unbox}\,x_2)(\mathsf{box}^+\,\iota.\,\mathsf{unbox}\,x)\\ &= \mathsf{box}[x \leftarrow \mathsf{box}^+\,\iota.\,\mathsf{unbox}\,x].\,\mathsf{case}\,x\,\mathsf{of}\,x_1.\,\mathsf{in}_1\,\mathsf{unbox}\,x_1\,;x_2.\,\mathsf{in}_2\,\mathsf{unbox}\,x_2\\ &= \mathsf{box}\,\iota.\,\mathsf{case}(\mathsf{box}^+\,\iota.\,\mathsf{unbox}\,x)\,\mathsf{of}\,x_1.\,\mathsf{in}_1\,\mathsf{unbox}\,x_1\,;x_2.\,\mathsf{in}_2\,\mathsf{unbox}\,x_2\\ &= \mathsf{box}\,\iota.\,\mathsf{case}(\mathsf{unbox}\,x)\,\mathsf{of}\,x_1.\,\mathsf{in}_1\,x_1\,;x_2.\,\mathsf{in}_2\,x_2\\ &= \mathsf{box}\,\iota.\,\mathsf{unbox}\,x\\ &= x\end{aligned}$$

The other direction requires the permutation of a substitution through box^+, the β-rule for $\blacksquare$, the commuting conversion of box^+ through case,

the reduction rule for box^+, and η-rules for $\blacksquare$ and sums:

$$\begin{aligned}
&(\lambda x.\, \mathsf{box}^+\, \iota.\, \mathsf{unbox}\, x)(\mathsf{box}\, \iota.\, \mathsf{case}\, x\, \mathsf{of}\, x_1.\, \mathsf{in}_1\, \mathsf{unbox}\, x_1; x_2.\, \mathsf{in}_2\, \mathsf{unbox}\, x_2)\\
&= \mathsf{box}^+[x \leftarrow \mathsf{box}\, \iota.\, \mathsf{case}\, x\, \mathsf{of}\, x_1.\, \mathsf{in}_1\, \mathsf{unbox}\, x_1; x_2.\, \mathsf{in}_2\, \mathsf{unbox}\, x_2].\, \mathsf{unbox}\, x\\
&= \mathsf{box}^+\, \iota.\, \mathsf{unbox}\, \mathsf{box}\, \iota.\, \mathsf{case}\, x\, \mathsf{of}\, x_1.\, \mathsf{in}_1\, \mathsf{unbox}\, x_1; x_2.\, \mathsf{in}_2\, \mathsf{unbox}\, x_2\\
&= \mathsf{box}^+\, \iota.\, \mathsf{case}\, x\, \mathsf{of}\, x_1.\, \mathsf{in}_1\, \mathsf{unbox}\, x_1; x_2.\, \mathsf{in}_2\, \mathsf{unbox}\, x_2\\
&= \mathsf{case}\, x\, \mathsf{of}\, x_1.\, \mathsf{box}^+\, \iota.\, \mathsf{in}_1\, \mathsf{unbox}\, x_1; x_2.\, \mathsf{box}^+\, \iota.\, \mathsf{in}_2\, \mathsf{unbox}\, x_2\\
&= \mathsf{case}\, x\, \mathsf{of}\, x_1.\, \mathsf{in}_1\, \mathsf{box}\, \iota.\, \mathsf{unbox}\, x_1; x_2.\, \mathsf{in}_2\, \mathsf{box}\, \iota.\, \mathsf{unbox}\, x_2\\
&= \mathsf{case}\, x\, \mathsf{of}\, x_1.\, \mathsf{in}_1\, x_1; x_2.\, \mathsf{in}_2\, x_2\\
&= x
\end{aligned}$$

As a final remark of this section, we note that our main direction of further work beyond this paper has been to extend the gλ-calculus with dependent types [21], as we will discuss further in Section 2.6.2. In this setting proofs take place inside the calculus, as with proof assistants such as Coq [63] and Agda [70]. The 'pen-and-paper' proofs of this section are therefore interesting partly because they reveal some of the constructions that are essential to proving properties of guarded recursive programs; these are the constructions that must be supported by the dependent type theory.

2.5 Behavioural Differential Equations

In this section we demonstrate the expressivity of the approach of this paper by showing how to construct coinductive streams as solutions to *behavioural differential equations* [78] in the gλ-calculus. This hence allows us to reason about such functions in $L\mathrm{g}\lambda$, instead of via bisimulation arguments.

2.5.1 Definition and Examples

We now define, and give examples of, behavioural differential equations. These examples will allow us to sketch informally how they can be expressed within the gλ-calculus, and how the program logic $L\mathrm{g}\lambda$ can be used to reason about them.

Definition 2.44. Let Σ be a first-order signature over a base sort A. A *behavioural differential equation* for a k-ary stream function is a pair of terms h_f and t_f (standing for *head* and *tail*), where h_f is a term containing function symbols from Σ, and variables as follows:

$$x_1, \ldots, x_k : A \vdash h_f : A$$

Intuitively, the variables x_i denote the heads of the argument stream. t_f is a term with function symbols from Σ along with a new constant f of sort $(\mathsf{Str}A)^k \to \mathsf{Str}A$, and variables as follows:

$$x_1, \ldots, x_k, y_1, \ldots, y_k, z_1, \ldots, z_k : \mathsf{Str}A \vdash t_f : \mathsf{Str}A$$

Intuitively, the variables x_i denote the streams whose head is the head of the argument stream and whose tails are all zeros, the variables y_i denote the argument streams, the variables z_i denote the tails of the argument streams, and the new constant f is recursive self-reference.

Further, given a set of stream functions defined by behavioural differential equations, the term t_f can use functions from that set as constants (behavioural differential equations are therefore *modular* in the sense of Milius et al. [65]).

Note that we have slightly weakened the original notion of behavioural differential equation by omitting the possibility of mutually recursive definitions, as used for example to define the stream of Fibonacci numbers [78, Section 5]. This omission will ease the notational burden involved in the formal results of the next section, but mutually recursive definitions can be accommodated within the gλ-calculus setting by, for example, considering a pair of mutually recursive stream functions as a function producing a pair of streams.

Example 2.45.

1. Assuming we have constant zero of type $\mathbf{N}$, the constant stream zeros of Example 2.9.4 is defined as a behavioural differential equation by

$$h_{\mathsf{zeros}} = \mathsf{zero} \qquad t_{\mathsf{zeros}} = \mathsf{zeros}$$

2. As an example of the modularity of this setting, given some $n : \mathbf{N}$ we can define the stream $[n]$ using the zeros stream defined above, by

$$h_{[n]} = n \qquad t_{[n]} = \mathsf{zeros}$$

3. Assuming we have addition $+ : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ written infix, then stream addition, also written $+$ and infix, is the binary function defined by

$$h_{+} = x_1 + x_2 \qquad t_{+} = z_1 + z_2$$

4. Assuming we have multiplication $\times : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$, written infix, then stream product, also written $\times$ and infix, is the binary function defined by

$$h_{\times} = x_1 \times x_2 \qquad t_{\times} = (z_1 \times y_2) + (x_1 \times z_2)$$

It is straightforward to translate the definitions above into constructions on guarded streams in the gλ-calculus. For example, stream addition is defined by the function on guarded streams $\mathsf{plus}^{\mathsf{g}} : \mathsf{Str}^{\mathsf{g}}\,\mathbf{N} \to \mathsf{Str}^{\mathsf{g}}\,\mathbf{N} \to \mathsf{Str}^{\mathsf{g}}\,\mathbf{N}$ below:

$$\mathsf{plus}^{\mathsf{g}} \triangleq \mathsf{fix}\,\lambda p.\lambda s_1.\lambda s_2.(\mathsf{hd}^{\mathsf{g}}\,s_1 + \mathsf{hd}^{\mathsf{g}}\,s_2) :: (p \circledast (\mathsf{tl}^{\mathsf{g}}\,s_1) \circledast (\mathsf{tl}^{\mathsf{g}}\,s_2))$$

We can lift this to a function on streams $\mathsf{plus} : \mathsf{Str}\,\mathbf{N} \to \mathsf{Str}\,\mathbf{N} \to \mathsf{Str}\,\mathbf{N}$ by $\mathsf{plus} \triangleq \mathcal{L}_2(\mathsf{plus}^{\mathsf{g}})$, recalling $\mathcal{L}_2$ from Example 2.43.2. Now by Lemma 2.40 we have

$$\mathsf{plus}^{\mathsf{g}} = \lambda s_1.\lambda s_2.(\mathsf{hd}^{\mathsf{g}}\, s_1 + \mathsf{hd}^{\mathsf{g}}\, s_2) :: ((\mathsf{next}\,\mathsf{plus}^{\mathsf{g}}) \circledast (\mathsf{tl}^{\mathsf{g}}\, s_1) \circledast (\mathsf{tl}^{\mathsf{g}}\, s_2)). \tag{2.11}$$

We can then prove in the logic $L\mathsf{g}\lambda$ that the definition of plus satisfies the specification given by the behavioural differential equation of Example 2.45.3. Given $s_1, s_2 : \mathsf{Str}\,\mathbf{N}$, we have

$$\begin{aligned}
\mathsf{hd}(\mathsf{plus}\, s_1\, s_2) &= \mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}(\mathsf{plus}\, s_1\, s_2) \\
&= \mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}(\mathcal{L}_2(\mathsf{plus}^{\mathsf{g}})\, s_1\, s_2) \\
&= \mathsf{hd}^{\mathsf{g}}(\mathsf{plus}^{\mathsf{g}}(\mathsf{unbox}\, s_1)(\mathsf{unbox}\, s_2)) && \text{(Example 2.43.2)} \\
&= (\mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}\, s_1) + (\mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}\, s_2) && \text{(2.11)} \\
&= (\mathsf{hd}\, s_1) + (\mathsf{hd}\, s_2)
\end{aligned}$$

For the tl case, that $\mathsf{tl}(\mathsf{plus}\, s_1\, s_2) = \mathsf{plus}(\mathsf{tl}\, s_1)(\mathsf{tl}\, s_2)$, we proceed similarly, but also using that $\mathsf{tl}^{\mathsf{g}}(\mathsf{unbox}\,\sigma) = \mathsf{next}(\mathsf{unbox}(\mathsf{tl}\,\sigma))$ which follows from the definition of tl, the β-rule for $\blacksquare$, and the η-rule for $\blacktriangleright$.

We can hence use $L\mathsf{g}\lambda$ to prove further properties of streams defined via behavioural differential equations, for example that stream addition is commutative. Such proofs proceed by conducting the proof on the guarded stream produced by applying unbox, then by introducing the $\Box$ modality so long as the context is suitably constant, and then by invoking Lemma 2.40.

2.5.2 From Behavioural Differential Equations to gλ-Terms

In the previous section we saw an example of a translation from a behavioural differential equation to a gλ-term. In this section we present the general translation. Starting with a k-ary behavioural differential equation (h_f, t_f) we will define a gλ-term[6]

$$\Phi^{\mathsf{g}}_f : \blacktriangleright\Big((\mathsf{Str}^{\mathsf{g}} A)^k \to \mathsf{Str}^{\mathsf{g}} A\Big) \to \Big((\mathsf{Str}^{\mathsf{g}} A)^k \to \mathsf{Str}^{\mathsf{g}} A\Big)$$

by induction on the structure of h_f and t_f. We may apply a fixed-point combinator to this to get a function on guarded streams, which we write as f^{g}.

We first extend gλ with function symbols in the signature Σ of (h_f, t_f). Using these it is straightforward to define a gλ term h^{g}_f of type

$$x_1 : A, x_2 : A, \cdots, x_k : A \vdash h^{\mathsf{g}}_f : A,$$

corresponding to h_f in the obvious way.

[6] We use the uncurried form to simplify the semantics.

From t_f we define the term t_f^{g} of type

$$\vec{x}, \vec{y} : \mathsf{Str}^{\mathrm{g}} A, \vec{z} : \blacktriangleright \mathsf{Str}^{\mathrm{g}} A, f : \blacktriangleright \Big((\mathsf{Str}^{\mathrm{g}} A)^k \to \mathsf{Str}^{\mathrm{g}} A \Big) \vdash t_f^{\mathrm{g}} : \blacktriangleright \mathsf{Str}^{\mathrm{g}} A$$

by induction on the structure of t_f as follows.

The base cases are simple:

- If $t_f = x_i$ for some i we put $t_f^{\mathrm{g}} = \mathsf{next}\, x_i$, and similarly for y_i;
- If $t_f = z_i$ we put $t_f^{\mathrm{g}} = z_i$.

If $t_f = f(a_1, \ldots, a_k)$ we put

$$t_f^{\mathrm{g}} = \mathrm{curry}^{\mathrm{g}}(f) \circledast t_{a_1}^{\mathrm{g}} \circledast \cdots \circledast t_{a_k}^{\mathrm{g}}$$

where $\mathrm{curry}^{\mathrm{g}}(f)$ is the currying of the function f, which is easily definable as a gλ term.

Finally if $t_f = e(a_1, \ldots, a_l)$ for some previously defined l-ary e then we put

$$t_f^{\mathrm{g}} = \mathrm{curry}^{\mathrm{g}}(\mathsf{next}\, e^{\mathrm{g}}) \circledast t_{a_1}^{\mathrm{g}} \circledast \cdots \circledast t_{a_l}^{\mathrm{g}}$$

We can then combine the terms h_f^{g} and t_f^{g} to define the desired term Φ_f^{g} as

$$\lambda f, \vec{y}. (h_f^{\mathrm{g}}[\mathsf{hd}^{\mathrm{g}}\, y_i / x_i]) :: (t_f^{\mathrm{g}}[(\mathsf{hd}^{\mathrm{g}}\, y_i :: \mathsf{next}\, \mathsf{zeros}) / x_i, \mathsf{tl}^{\mathrm{g}}\, y_i / z_i])$$

Analogously from a behavioural differential equation we define a gλ term Φ_f of type

$$\Phi_f : \Big((\mathsf{Str} A)^k \to \mathsf{Str} A \Big) \to \Big((\mathsf{Str} A)^k \to \mathsf{Str} A \Big),$$

where for the function symbols we take the lifted (as in Example 2.43.2) function symbols used in the definition of Φ_f^{g}.

We will now show that the lifting of the unique fixed point of Φ_f^{g} is a fixed point of Φ_f, and hence satisfies the behavioural differential equation for f. We prove this using denotational semantics, relying on its adequacy (Corollary 2.31).

2.5.3 The Topos of Trees as a Sheaf Category

In order to reach the formal results regarding behavioural differential equations of the next section, it will be convenient to provide an alternative definition for the topos of trees as a category of *sheaves*, rather than *presheaves*.

The preorder $\omega = 1 \leq 2 \leq \cdots$ is a topological space given the *Alexandrov topology* where the open sets are the *downwards* closed sets. These downwards closed sets are simply $0 \subseteq 1 \subseteq 2 \subseteq \cdots \subseteq \omega$, where 0 is the empty set, n is the

downwards closure of n for any positive integer n, and ω is the entire set. Then the sheaves X over this topological space, $\mathrm{Sh}(\omega)$, are presheaves over these open sets obeying certain properties [61]. In this case these properties ensure that $X(0)$ must always be a singleton set and $X(\omega)$ is entirely determined (up to isomorphism) by the sets $X_1, X_2, \cdots$ as their *limit*. This definition is hence plainly equivalent to the definition of $\mathcal{S}$ from Section 2.3.1.

However this presentation is more convenient for our purposes here, in which we will need to go back and forth between the categories $\mathcal{S}$ and **Set**, because the global sections functor[7] Γ in the sequence of adjoints

$$\Pi_1 \dashv \Delta \dashv \Gamma$$

where

$$\begin{array}{ccc} \Pi_1 : \mathcal{S} \to \mathbf{Set} & \Delta : \mathbf{Set} \to \mathcal{S} & \Gamma : \mathcal{S} \to \mathbf{Set} \\ \Pi_1(X) = X(1) & \Delta(a)(\alpha) = \begin{cases} 1 & \text{if } \alpha = 0 \\ a & \text{otherwise} \end{cases} & \Gamma(X) = X(\omega) \end{array}$$

is just evaluation at ω, i.e. the limit is already present, which simplifies notation. Another advantage is that $\blacktriangleright : \mathcal{S} \to \mathcal{S}$ is given as

$$\begin{aligned} (\blacktriangleright X)(\nu + 1) &= X(\nu) \\ (\blacktriangleright X)(\alpha) &= X(\alpha) \end{aligned}$$

where α is a limit ordinal (either 0 or ω) which means that $\blacktriangleright X(\omega) = X(\omega)$ and as a consequence, $\mathbf{next}_\omega = \mathrm{id}_{X(\omega)}$ and $\Gamma(\blacktriangleright X) = \Gamma(X)$ for any $X \in \mathcal{S}$ and so $\blacksquare(\blacktriangleright X) = \blacksquare X$ for any X, so we do not have to deal with mediating isomorphisms.

We finally turn to a useful lemma which we will use in the next section.

Lemma 2.46. *Let X, Y be objects of $\mathcal{S}$. Let $F : \blacktriangleright\left(Y^X\right) \to Y^X$ be a morphism in $\mathcal{S}$ and $\underline{F} : Y(\omega)^{X(\omega)} \to Y(\omega)^{X(\omega)}$ be a* function *in* **Set**. *Suppose that the diagram*

$$\begin{array}{ccc} \Gamma\left(\blacktriangleright\left(Y^X\right)\right) & \xrightarrow{\Gamma(F)} & \Gamma(Y^X) \\ \Big\downarrow{\scriptstyle \lim} & & \Big\downarrow{\scriptstyle \lim} \\ Y(\omega)^{X(\omega)} & \xrightarrow{\underline{F}} & Y(\omega)^{X(\omega)} \end{array}$$

[7] The standard notation Γ for this functor should not be confused with our notation for typing contexts.

commutes, where $\lim\left(\{g_\nu\}_{\nu=0}^{\omega}\right) = g_\omega$. *By Banach's fixed point theorem* F *has a unique fixed point, say* $u : 1 \to Y^X$.

Then $\lim(\Gamma(u)(*)) = \lim(\Gamma(\mathbf{next} \circ u)(*)) = \Gamma(\mathbf{next} \circ u)(*)_\omega = u_\omega(*)_\omega$ *is a fixed point of* $\underline{F}$.

Proof.

$$\begin{aligned}\underline{F}(\lim(\Gamma(u)(*))) &= \lim(\Gamma(F)(\Gamma(\mathbf{next} \circ u)(*))) \\ &= \lim(\Gamma(F \circ \mathbf{next} \circ u)(*)) = \lim(\Gamma(u)(*)).\end{aligned}$$

□

Note that lim is not an isomorphism, as there are in general many more functions from $X(\omega)$ to $Y(\omega)$ than those that arise from natural transformations. The ones that arise from natural transformations are the *non-expansive* ones.

2.5.4 Expressing Behavioural Differential Equations

We first define two interpretations of behavioural differential equations (Definition 2.44); first in the topos of trees, and then in **Set**. The interpretation in $\mathcal{S}$ is just the denotation of the term Φ_f^{g} from Section 2.5.2, whereas the inclusion of the interpretation in **Set** into the topos of trees, using the constant presheaf functor Δ, is the denotation of the term Φ_f from Section 2.5.2.

Definition 2.47. Fixing a set $|A|$ which will interpret our base sort, define $[\![A]\!]_{\mathcal{S}} = \Delta|A|$ and $[\![\mathsf{Str}A]\!]_{\mathcal{S}} = \mu X.\Delta|A| \times \blacktriangleright X$; that is, the denotation of $\mathrm{Str}^{\mathrm{g}}(\Delta|A|)$ from Example 2.16.1. To each function symbol $g \in \Sigma$ of type $\tau_1, \ldots, \tau_n \to \tau_{n+1}$ we assign a morphism

$$[\![g]\!]_{\mathcal{S}} : [\![\tau_1]\!]_{\mathcal{S}} \times [\![\tau_2]\!]_{\mathcal{S}} \times \cdots \times [\![\tau_n]\!]_{\mathcal{S}} \to [\![\tau_{n+1}]\!]_{\mathcal{S}}.$$

We then interpret h_f as a morphism of type $[\![A]\!]_{\mathcal{S}}^k \to [\![A]\!]_{\mathcal{S}}$ by induction:

$$\begin{aligned}[\![x_i]\!]_{\mathcal{S}} &= \pi_i \\ [\![g(t_1, t_2, \ldots, t_n)]\!]_{\mathcal{S}} &= [\![g]\!]_{\mathcal{S}} \circ \langle [\![t_1]\!]_{\mathcal{S}}, [\![t_2]\!]_{\mathcal{S}}, \cdots, [\![t_n]\!]_{\mathcal{S}} \rangle.\end{aligned}$$

t_f will be interpreted similarly, but we also have the new function symbol f to consider. The interpretation of t_f is therefore a $\mathcal{S}$-arrow of type

$$\left[\!\left[t_f\right]\!\right]_{\mathcal{S}} : [\![\mathsf{Str}A]\!]_{\mathcal{S}}^k \times [\![\mathsf{Str}A]\!]_{\mathcal{S}}^k \times (\blacktriangleright([\![\mathsf{Str}A]\!]_{\mathcal{S}}))^k \times \blacktriangleright\left([\![\mathsf{Str}A]\!]_{\mathcal{S}}^{[\![\mathsf{Str}A]\!]_{\mathcal{S}}^k}\right) \to \blacktriangleright([\![\mathsf{Str}A]\!]_{\mathcal{S}})$$

and is defined as:

$$\begin{aligned}
[\![x_i]\!]_{\mathcal{S}} &= \mathbf{next} \circ \pi_{x_i} \\
[\![y_i]\!]_{\mathcal{S}} &= \mathbf{next} \circ \pi_{y_i} \\
[\![z_i]\!]_{\mathcal{S}} &= \pi_{z_i} \\
[\![g(t_1, t_2, \ldots, t_n)]\!]_{\mathcal{S}} &= \blacktriangleright([\![g]\!]_{\mathcal{S}}) \circ \mathbf{can} \circ \langle [\![t_1]\!]_{\mathcal{S}}, [\![t_2]\!]_{\mathcal{S}}, \cdots, [\![t_n]\!]_{\mathcal{S}} \rangle \qquad \text{if } g \neq f \\
[\![f(t_1, t_2, \ldots, t_k)]\!]_{\mathcal{S}} &= \mathbf{eval} \circ \left\langle J \circ \pi_f, \mathbf{can} \circ \langle [\![t_1]\!]_{\mathcal{S}}, [\![t_2]\!]_{\mathcal{S}}, \cdots, [\![t_k]\!]_{\mathcal{S}} \rangle \right\rangle
\end{aligned}$$

where **can** is the canonical isomorphism witnessing that $\blacktriangleright$ preserves products; **eval** is the evaluation map, and J is the map $\blacktriangleright(X \to Y) \to \blacktriangleright X \to \blacktriangleright Y$ which gives $\blacktriangleright$ its applicative functor structure $\circledast$.

We can then define the $\mathcal{S}$-arrow

$$F : \blacktriangleright\left([\![\mathsf{Str}A]\!]_{\mathcal{S}}^{[\![\mathsf{Str}A]\!]_{\mathcal{S}}^k} \right) \to [\![\mathsf{Str}A]\!]_{\mathcal{S}}^{[\![\mathsf{Str}A]\!]_{\mathcal{S}}^k}$$

as the exponential transpose of

$$F' = \mathbf{fold} \circ \left\langle \left[\!\!\left[h_f \right]\!\!\right] \circ \vec{\mathbf{hd}} \circ \pi_1, \left[\!\!\left[t_f \right]\!\!\right]_{\mathcal{S}} \circ \left(\left\langle \iota \circ \vec{\mathbf{hd}}, \mathrm{id}_{[\![\mathsf{Str}A]\!]_{\mathcal{S}}^k}, \vec{\mathbf{tail}} \right\rangle \times \mathrm{id}_{\blacktriangleright\left([\![\mathsf{Str}A]\!]_{\mathcal{S}}^{[\![\mathsf{Str}A]\!]_{\mathcal{S}}^k} \right)} \right) \right\rangle$$

where **hd** and **tl** are head and tail functions, extended in the obvious way to tuples, and $\mathbf{fold} : [\![A]\!]_{\mathcal{S}} \times \blacktriangleright [\![\mathsf{Str}A]\!]_{\mathcal{S}} \to [\![\mathsf{Str}A]\!]_{\mathcal{S}}$ is the evident 'cons' arrow. The function ι maps an element in A to the guarded stream with head a and tail the stream of zeroes.

Definition 2.48. We now use the topos of trees definition above to define the denotation of h_f and t_f in **Set**. We set $[\![A]\!]_{\mathbf{Set}} = |A|$ and $[\![\mathsf{Str}A]\!]_{\mathbf{Set}} = [\![\mathsf{Str}A]\!]_{\mathcal{S}}(\omega)$. For each function symbol in Σ we define $[\![g]\!]_{\mathbf{Set}} = \Gamma[\![g]\!]_{\mathcal{S}} = \left([\![g]\!]_{\mathcal{S}} \right)_{\omega}$.

We then define $\left[\!\!\left[h_f \right]\!\!\right]_{\mathbf{Set}}$ as a function

$$[\![A]\!]_{\mathbf{Set}}^k \to [\![A]\!]_{\mathbf{Set}}$$

exactly as we defined $\left[\!\!\left[h_f \right]\!\!\right]_{\mathcal{S}}$:

$$\begin{aligned}
[\![x_i]\!]_{\mathbf{Set}} &= \pi_i \\
[\![g(t_1, t_2, \ldots, t_n)]\!]_{\mathbf{Set}} &= [\![g]\!]_{\mathbf{Set}} \circ \langle [\![t_1]\!]_{\mathbf{Set}}, [\![t_2]\!]_{\mathbf{Set}}, \cdots, [\![t_n]\!]_{\mathbf{Set}} \rangle .
\end{aligned}$$

The denotation of t_f is somewhat different, as we do not have the functor $\blacktriangleright$. We define

$$\left[\!\!\left[t_f \right]\!\!\right]_{\mathbf{Set}} : [\![A]\!]_{\mathbf{Set}}^k \times [\![\mathsf{Str}A]\!]_{\mathbf{Set}}^k \times [\![\mathsf{Str}A]\!]_{\mathbf{Set}}^k \times [\![\mathsf{Str}A]\!]_{\mathbf{Set}}^{[\![\mathsf{Str}A]\!]_{\mathbf{Set}}^k} \to [\![\mathsf{Str}A]\!]_{\mathbf{Set}}$$

as follows:

$$\begin{aligned}
[\![x_i]\!]_{\mathbf{Set}} &= \pi_{x_i} \\
[\![y_i]\!]_{\mathbf{Set}} &= \pi_{y_i} \\
[\![z_i]\!]_{\mathbf{Set}} &= \pi_{z_i} \\
[\![g(t_1, t_2, \ldots, t_n)]\!]_{\mathbf{Set}} &= [\![g]\!]_{\mathbf{Set}} \circ \langle [\![t_1]\!]_{\mathbf{Set}}, [\![t_2]\!]_{\mathbf{Set}}, \cdots, [\![t_n]\!]_{\mathbf{Set}} \rangle \qquad \text{if } g \neq f \\
[\![f(t_1, t_2, \ldots, t_k)]\!]_{\mathbf{Set}} &= \mathbf{eval} \circ \left\langle \pi_f, \langle [\![t_1]\!]_{\mathbf{Set}}, [\![t_2]\!]_{\mathbf{Set}}, \cdots, [\![t_k]\!]_{\mathbf{Set}} \rangle \right\rangle.
\end{aligned}$$

We then define

$$\underline{F} : [\![\mathsf{Str}A]\!]_{\mathbf{Set}}^{[\![\mathsf{Str}A]\!]_{\mathbf{Set}}^k} \to [\![\mathsf{Str}A]\!]_{\mathbf{Set}}^{[\![\mathsf{Str}A]\!]_{\mathbf{Set}}^k}$$

as

$$\underline{F}(\varphi)(\vec{\sigma}) = \Gamma(\mathbf{fold})\left(\left[\!\!\left[h_f \right]\!\!\right]_{\mathbf{Set}} \circ \vec{\mathbf{hd}}(\vec{\sigma}), \left[\!\!\left[t_f \right]\!\!\right]_{\mathbf{Set}} \left(\iota\left(\vec{\mathbf{hd}}(\vec{\sigma}) \right), \vec{\sigma}, \vec{\mathbf{tl}}(\vec{\sigma}), \varphi \right) \right)$$

Lemma 2.49. *For the above defined F and $\underline{F}$ we have*

$$\mathsf{lim} \circ \Gamma(F) = \underline{F} \circ \mathsf{lim}.$$

Proof. Take $\varphi \in \Gamma\left(\blacktriangleright \left([\![\mathsf{Str}A]\!]_{\mathcal{S}}^{[\![\mathsf{Str}A]\!]_{\mathcal{S}}^k} \right) \right) = \Gamma\left([\![\mathsf{Str}A]\!]_{\mathcal{S}}^{[\![\mathsf{Str}A]\!]_{\mathcal{S}}^k} \right)$. We have

$$\mathsf{lim}(\Gamma(F)(\varphi)) = \mathsf{lim}\,(F_\omega(\varphi)) = F_\omega(\varphi)_\omega$$

and

$$\underline{F}(\mathsf{lim}(\varphi)) = \underline{F}(\varphi_\omega)$$

These are both elements of $[\![\mathsf{Str}A]\!]_{\mathbf{Set}}^{[\![\mathsf{Str}A]\!]_{\mathbf{Set}}^k}$, and so are functions in **Set**, so to show they are equal we can use elements. Take $\vec{\sigma} \in [\![\mathsf{Str}A]\!]_{\mathbf{Set}}^k$. We are then required to show

$$\underline{F}(\varphi_\omega)(\vec{\sigma}) = F_\omega(\varphi)_\omega(\vec{\sigma})$$

Recall that F is the exponential transpose of F', so $F_\omega(\varphi)_\omega(\vec{\sigma}) = F'_\omega(\varphi, \vec{\sigma})$. Now recall that composition in $\mathcal{S}$ is just composition of functions at each stage, that products in $\mathcal{S}$ are defined pointwise, and that $\mathbf{next}_\omega$ is the identity function. Moreover, the $\mathcal{S}$-arrow **hd** gets mapped by Γ to **hd** in **Set** and the same holds for **tl**. For the latter it is important that $\Gamma(\blacktriangleright(X)) = \Gamma(X)$ for any X.

We thus get

$$F'_\omega(\varphi, \vec{\sigma}) = \mathbf{fold}_\omega \left((\left[\!\!\left[h_f \right]\!\!\right]_{\mathcal{S}})_\omega(\mathbf{hd}(\vec{\sigma})), \left(\left[\!\!\left[t_f \right]\!\!\right]_{\mathcal{S}} \right)_\omega (\varphi, \iota(\mathbf{hd}(\vec{\sigma})), \vec{\sigma}, \mathbf{tl}(\vec{\sigma})) \right)$$

and also

$$\underline{F}(\varphi_\omega)(\vec{\sigma}) = \mathbf{fold}_\omega\left(\left[\!\left[h_f\right]\!\right]_{\mathbf{Set}}(\mathbf{hd}(\vec{\sigma})), \left(\left[\!\left[t_f\right]\!\right]_{\mathbf{Set}}\right)(\varphi_\omega, \iota(\mathbf{hd}(\vec{\sigma})), \vec{\sigma}, \mathbf{tl}(\vec{\sigma}))\right)$$

It is now easy to see that these two are equal, by induction on the structure of h_f and t_f. The variable cases are trivial, but crucially use the fact that $\mathbf{next}_\omega$ is the identity. The cases for function symbols in Σ are trivial by the definition of their denotations in **Set**. The case for f goes through similarly since application at ω only uses φ at ω. □

Theorem 2.50. *Let Σ be a signature and suppose we have an interpretation in $\mathcal{S}$. Let (h_f, t_f) be a behavioural differential equation defining a k-ary function f using function symbols in Σ. The right-hand sides of h_f and t_f define a gλ-term Φ_f^{g} of type*

$$\Phi_f^{\mathrm{g}} : \blacktriangleright\left(\mathsf{Str}^{\mathrm{g}}\,\mathbf{N}^k \to \mathsf{Str}^{\mathrm{g}}\,\mathbf{N}\right) \to \left(\mathsf{Str}^{\mathrm{g}}\,\mathbf{N}^k \to \mathsf{Str}^{\mathrm{g}}\,\mathbf{N}\right)$$

and a term Φ_f of type

$$\Phi_f : \blacktriangleright\left(\mathsf{Str}\,\mathbf{N}^k \to \mathsf{Str}\,\mathbf{N}\right) \to \left(\mathsf{Str}\,\mathbf{N}^k \to \mathsf{Str}\,\mathbf{N}\right)$$

(here we must 'lift' the interpretations of the function symbols in Σ from guarded recursive streams to coinductive streams; this can be done by analogy with the $\mathcal{L}$ functions of Example 2.43.2.)

Let $f^{\mathrm{g}} = \mathsf{fix}\,\Phi_f^{\mathrm{g}}$ be the fixed point of Φ_f^{g}. Then $f = \mathcal{L}_k(\mathsf{box}\, f^{\mathrm{g}})$ is a fixed point of Φ_f which in turn implies that it satisfies equations h_f and t_f.

Proof. The morphism F in Lemma 2.46 is the interpretation of the term Φ_f^{g} from Section 2.5.2. The inclusion of the morphism $\underline{F}$ in Lemma 2.46 is the denotation of the term Φ_f. Further, the inclusion (with Δ) of the fixed point constructed in Lemma 2.46 is the denotation of f.

Proposition 2.49 concludes the proof that $[\![f]\!]$ is indeed a fixed point of $[\![\Phi_f]\!]$. Hence by adequacy of the denotational semantics we have that f is a fixed point of Φ_f. □

This concludes our proof that for each behavioural differential equation that defines a function on streams, we can use the gλ-calculus to define its solution.

2.6 Concluding Remarks

We have seen how the guarded lambda-calculus, or gλ-calculus, allows us to program with guarded recursive and coinductive data structures while retaining normalisation and productivity, and how the topos of trees provides adequate semantics and an internal logic L gλ for reasoning about gλ-programs. We have demonstrated our approach's expressivity by showing that it can

express behavioural differential equations, a well-known format for the definition of stream functions. We conclude by surveying some related work and discussing some future directions.

2.6.1 Related Work

Other Calculi with Later. Since Nakano's original paper [69] there have been a number of calculi presented that utilise the later modality. Many of these calculi are *causal* [3, 54–57, 75, 79], in that they cannot express acausal but productive functions, and are therefore less expressive in this respect than the guarded λ-calculus. Note that this restriction is a feature, rather than a defect, for some applications such as functional reactive programming [56], where programs should indeed be prevented from reacting to an event before it has occurred. We could similarly program in the fragment of the gλ-calculus without $\blacksquare$ to retain this guarantee. We further note that the gλ-calculus is intended to extend the simply typed λ-calculus in as modest a way as possible while gaining the expressivity we desire, and so we have avoided exotic features such as Nakano's subtyping and first-class type equalities (which make type inference a non-trivial open problem [77, Section 9]), or the use of natural numbers to stratify typing judgments [56], or reduction [3].

Atkey and McBride's clock quantifiers [8] showed how to express *acausal* functions in a calculus with later. This was extended to dependent types by Møgelberg [66], with improvements made subsequently by Bizjak and Møgelberg [19]. However the conference version of this paper [27] is the first to present operational semantics for such a calculus.

Clock quantifiers differ in two main ways from this paper's use of the modality $\blacksquare$. First, multiple clocks are useful for expressing nested coinductive types that intuitively vary on multiple independent time streams, such as infinite-breadth infinite-depth trees. We conjecture that we could accommodate this by extending our calculus with multiple versions of our type- and term-formers: $\mu^\kappa, \blacktriangleright^\kappa, \blacksquare^\kappa, \mathsf{next}^\kappa$ and so forth, labelled by clocks κ. Guardedness and constantness side-conditions on type- and term-formation would then check only the clock under consideration. Semantics could be given via presheaves over ω^n, where n is the number of clocks. One slightly awkward note is that we appear to need a new term-former to construct the isomorphism $\blacksquare^\kappa \blacktriangleright^{\kappa'} A \to \blacktriangleright^{\kappa'} \blacksquare^\kappa A$, given as a first-class type equality by Atkey and McBride [8] (the other direction of this isomorphism, and the permutation of $\blacksquare^\kappa$ with $\blacksquare^{\kappa'}$, are readily definable as terms).

Second, and more importantly, clock quantifiers remove the need for term-formers such as box to carry explicit substitutions. There is no free lunch however, as we must instead handle side-conditions asserting that given clock variables are free in the clock context; while such 'freshness' conditions are common in formal calculi they are a notorious source of error when reasoning about syntax. Further, if explicit substitutions are to be completely avoided

the prev constructor needs to be reworked, for example by replacing it with a force term-former [8], and so we no longer have a conventional destructor for ▶, so $\beta\eta$-equalities become more complex. Reiterating our remarks of Section 2.2.1 we note that, with respect to programming with the gλ-calculus, the burden presented by the explicit substitutions seems quite small, as all example programs involve identity substitutions only. Therefore our use of the ■ modality seems the simpler choice, especially as it allows us to adapt previously published work on term calculus for the modal logic Intuitionistic S4 [9]. However in our work on extending guarded type theory to dependent types [21] the explicit substitutions become more burdensome, resulting in our adoption of clock quantifiers for that work.

Dual Contexts. Our development draws extensively on the term calculus for Intuitionistic S4 of Bierman and de Paiva [9]. Subsequent work by Davies and Pfenning [35] modified Bierman and de Paiva's calculus, removing the explicit substitutions attached to the box term-former. As ever there is no free lunch, as instead a 'dual context' is used – the variable context has two compartments, one of which is reserved for constant types. The calculus is then closed under substitution via a modification of the definition of substitution to depend on which context the variable is drawn from. Because, as stated above, we found the burden of explicit substitutions not so great, we preferred to use the Bierman-de Paiva calculus as our basis rather than deal with this more complicated notion of substitution; however from our point of view these differences are relatively marginal and largely a matter of taste.

Ultrametric Spaces. As noted in the proof of Lemma 2.38, the category $\mathcal{M}$ of bisected complete non-empty ultrametric spaces is a complete subcategory of the topos of trees, corresponding to the total and inhabited $\mathcal{S}$-objects [14, Section 5]. This category $\mathcal{M}$ was shown to provide semantics for Nakano's calculus by Birkedal et al. [11], as well as for a related calculus with later by Krishnaswami and Benton [56]. These works do not feature the ■ modality, but its definition is easy - it maps any space to the space with the same underlying set, but the discrete metric. Why, then, do we instead use the topos of trees? First, $\mathcal{M}$ is not a topos, and therefore our work reasoning with the internal logic would not be possible. Second, $\mathcal{M}$ contains only *non-empty* spaces and so cannot model the **0** type. If the empty space is added then ▶ becomes undefinable: either ▶0 has underlying set $\emptyset$, in which case there exists a map $\blacktriangleright 0 \to 0$ and so the fixpoint function $(\blacktriangleright 0 \to 0) \to 0$ cannot exist without creating an inhabitant of 0, or the underlying set is not empty, in which case there is no map $\blacksquare\blacktriangleright 0 \to \blacksquare 0$, and so we cannot eliminate ▶ in constant contexts.

Sized Types. The best developed type-based method for ensuring productivity are sized types, introduced by Hughes et al. in 1996 [45]. They have now been implemented in the proof assistant Agda, following work by Abel [1]. There is as yet no equivalent development employing the later modality, so direct comparison on realistic examples with respect to criteria such as ease of use are probably premature. However we can make some preliminary observations. First, defining denotational semantics in a topos was essential to the development of the program logic $Lg\lambda$; to our knowledge there is no semantics of sized types yet developed that would support a similar development. Second, the later modality has applications that appear quite unrelated to sized types, in particular for modelling and reasoning about programming languages, starting with Appel et al. [7] and including, for example, the program logic iCAP [82]. These applications require recursive types with negative occurrences of the recursion variable, and so lie outside the scope of sized types. The implementation of guarded recursive types directly in a proof assistant should support such applications. Here the most relevant comparison will be with the Coq formalisations of semantics for later (in these cases, ultrametric semantics) [52, 80] as a basis for program logics. The Coq formalisation of the topos of trees via 'forcing' of Jaber et al. [46] may also be useable for such reasoning. Our hope is that implementing guarded recursive types as primitive might reduce the overhead involved in working indirectly on encoded semantics.

Similar Type- and Term-Formers We finally mention two further constructions that bear some resemblance to those of this paper. First, the ∞ type-former, and 'delay' $\sharp$, and 'force' $\flat$ type-formers, for coinduction in Agda [34, Section 2.2], look somewhat like $\blacktriangleright$, next, and prev respectively, but are not intended to replace syntactic guardedness checking and so the resemblance is largely superficial. Second, the 'next' and 'globally' modalities of (discrete time) Linear Temporal Logic, recently employed as type-formers for functional reactive programming by Jeltsch [50] and Jeffrey [49], look somewhat like $\blacktriangleright$ and $\blacksquare$, but we as yet see no obvious formal links between these approaches.

2.6.2 Further Work

Dependent Types. As discussed earlier, a major goal of this research is to extend the simply-typed $g\lambda$-calculus to a calculus with dependent types. This could provide a basis for interactive theorem proving with the later modality, integrating the sorts of proofs we performed in Section 2.4 into the calculus itself. In Bizjak et al. [21] we have developed an extensional guarded dependent type theory, which is proved sound in a model based on the topos of trees. This extension is not entirely straightforward, most notably requiring novel constructions to generalise applicative functor structure to

dependent types. The next challenge is to develop a type theory with decidable type checking, which would provide a basis for implementation. We have developed a type theory with later [15] based on *cubical type theory* [29], which has a notion of path equality which seems to interact better with the new constructs of guarded type theory than the ordinary Martin-Löf identity type. We conjecture that our new type theory has decidable type checking, but this property is still open even for cubical type theory without guarded recursion.

Inference of* gλ *Type- and Term-Formers. The examples in this paper make clear that programming in the gλ-calculus is usually a matter of 'decorating' conventional programs with our novel type- and term-formers such as $\blacktriangleright$ and next. This decoration process is often straightforward, but we are not insensitive to the burden on the programmer of demanding large amounts of novel notation be applied to their program before it will type-check. It would therefore be helpful to investigate algorithmic support for automatically performing this decoration process.

Full Abstraction. Corollary 2.31 established the soundness of our denotational semantics with respect to contextual equivalence. Its converse, full abstraction, is left open. A proof of full abstraction, or a counter-example, would help us to understand how good a model the topos of trees provides for the gλ-calculus, with respect to whether it differentiates terms that are operationally equivalent. Conversely, if full abstraction were found to fail we could ask whether a language extension is possible which brings the gλ-calculus closer to its intended semantics.

Acknowledgements

We gratefully acknowledge our discussions with Andreas Abel, Robbert Krebbers, Tadeusz Litak, Stefan Milius, Rasmus Møgelberg, Filip Sieczkowski, Bas Spitters, and Andrea Vezzosi, and the comments of the anoymous reviewers of both this paper and its conference version. 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 is supported in part by a Microsoft Research PhD grant.

Chapter 3

Dependent Types

This chapter consists of the paper:

[22] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal.
Guarded dependent type theory with coinductive types.
In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 20–35, 2016.

along with a technical appendix.

Abstract

We present guarded dependent type theory, GDTT, an extensional dependent type theory with a 'later' modality and clock quantifiers for programming and proving with guarded recursive and coinductive types. The later modality is used to ensure the productivity of recursive definitions in a modular, type based, way. Clock quantifiers are used for controlled elimination of the later modality and for encoding coinductive types using guarded recursive types. Key to the development of GDTT are novel type and term formers involving what we call 'delayed substitutions'. These generalise the applicative functor rules for the later modality considered in earlier work, and are crucial for programming and proving with dependent types. We show soundness of the type theory with respect to a denotational model.

3.1 Introduction

Dependent type theory is useful both for programming, and for proving properties of elements of types. Modern implementations of dependent type theories such as Coq [63], Nuprl [30], Agda [70], and Idris [23], have been used successfully in many projects. However, they offer limited support for programming and proving with *coinductive* types.

One of the key challenges is to ensure that functions on coinductive types are well-defined; that is, productive with unique solutions. Syntactic guarded recursion [31], as used for example in Coq [41], ensures productivity by requiring that recursive calls be nested directly under a constructor, but it is well known that such syntactic checks exclude many valid definitions, particularly in the presence of higher-order functions.

To address this challenge, a *type-based* approach to guarded recursion, more flexible than syntactic checks, was first suggested by Nakano [69]. A new modality, written $\triangleright$ and called 'later' [7], allows us to distinguish between data we have access to now, and data which we will get later. This modality must be used to guard self-reference in type definitions, so for example *guarded streams* of natural numbers are described by the guarded recursive equation

$$\mathsf{Str}^g_{\mathbb{N}} \simeq \mathbb{N} \times \triangleright \mathsf{Str}^g_{\mathbb{N}}$$

asserting that stream heads are available now, but tails only later.

Types defined via guarded recursion with $\triangleright$ are not standard coinductive types, as their denotation is defined via models based on the *topos of trees* [14]. More pragmatically, the bare addition of $\triangleright$ disallows productive but *acausal* [56] functions such as the 'every other' function that returns every second element of a stream. Atkey and McBride proposed *clock quantifiers* [8] for such functions; these have been extended to dependent types [19, 66], and Møgelberg [66, Thm. 2] has shown that they allow the definition of types whose denotation is precisely that of standard coinductive types interpreted in set-based semantics. As such, they allow us to program with real coinductive types, while retaining productivity guarantees.

In this paper we introduce the extensional guarded dependent type theory GDTT, which provides a framework where guarded recursion can be used not just for programming with coinductive types but also for coinductive reasoning.

As types depend on terms, one of the key challenges in designing GDTT is coping with elements that are only available later, i.e., elements of types of the form $\triangleright A$. We do this by generalising the applicative functor structure of $\triangleright$ to the dependent setting. Recall the rules for applicative functors [64]:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \triangleright A} \qquad \frac{\Gamma \vdash f : \triangleright(A \to B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash f \circledast t : \triangleright B} \tag{3.1}$$

The first rule allows us to make later use of data that we have now. The second allows, for example, functions to be applied recursively to the tails of streams.

Suppose now that f has type $\triangleright(\Pi x : A.B)$, and t has type $\triangleright A$. What should the type of $f \circledast t$ be? Intuitively, t will eventually reduce to some value $\mathsf{next}\, u$, and so the resulting type should be $\triangleright(B[u/x])$, but if t is an open term we may not be able to perform this reduction. This problem occurs in coinductive reasoning: if, e.g., A is $\mathsf{Str}^g_{\mathbb{N}}$, and B a property of streams, in our applications

f will be a (guarded) coinduction assumption that we will want to apply to the tail of a stream, which has type $\triangleright \mathsf{Str}^g_{\mathbb{N}}$.

We hence must introduce a new notion, of *delayed substitution*, similar to let-binding, allowing us to give $f \circledast t$ the type

$$\triangleright[x \leftarrow t].B$$

binding x in B. Definitional equality rules then allow us to simplify this type when t has form $\mathsf{next}\, u$, i.e., $\triangleright[x \leftarrow \mathsf{next}\, u].B \equiv \triangleright(B[u/x])$. This construction generalises to bind a list of variables. Delayed substitution is essential to many examples, as shown in Sec. 3.3, and surprisingly the applicative functor term-former $\circledast$, so central to the standard presentation of applicative functors, turns out to be *definable* via delayed substitutions, as shown in Sec. 3.2.

Contributions. The contributions of this paper are:

- We introduce the extensional guarded dependent type theory GDTT, and show that it gives a framework for programming and proving with guarded recursive and coinductive types. The key novel feature is the generalisation of the 'later' type-former and 'next' term-former via *delayed substitutions*;
- We prove the soundness of GDTT via a model similar to that used in earlier work on guarded recursive types and clock quantifiers [19, 66].

We focus on the design and soundness of the type theory and restrict attention to an extensional type theory. We postpone a treatment of an intensional version of the theory to future work (see Secs. 3.7 and 3.8).

In addition to the examples included in this paper, we are pleased to note that a preliminary version of GDTT has already proved crucial for formalizing a logical relations adequacy proof of a semantics for PCF using guarded recursive types by Paviotti et. al. [72].

3.2 Guarded Dependent Type Theory

GDTT is a type theory with base types unit $\mathbf{1}$, booleans $\mathbf{B}$, and natural numbers N, along with Π-types, Σ-types, identity types, and universes. For space reasons we omit all definitions that are standard to such a type theory; see e.g. Jacobs [47]. Our universes are à la Tarski, so we distinguish between types and terms, and have terms that represent types; they are called *codes* of types and they can be recognised by their circumflex, e.g., $\widehat{\mathsf{N}}$ is the code of the type N. We have a map El sending codes of types to their corresponding type. We follow standard practice and often omit El in examples, except where it is important to avoid confusion.

We fix a countable set of *clock variables* $\mathrm{CV} = \{\kappa_1, \kappa_2, \cdots\}$ and a single *clock constant* κ_0, which will be necessary to define, for example, the function hd in Sec. 3.5. A *clock* is either a clock variable or the clock constant; they are intuitively temporal dimensions on which types may depend. A *clock context* $\Delta, \Delta', \cdots$ is a finite *set* of *clock variables*. We use the judgement $\vdash_\Delta \kappa$ to express that either κ is a clock variable in the set Δ or κ is the clock constant κ_0. All judgements, summarised in Fig. 3.1, are parametrised by clock contexts. Codes of types inhabit *universes* $\mathcal{U}_\Delta$ parametrised by clock contexts similarly. The universe $\mathcal{U}_\Delta$ is only well-formed in clock contexts Δ' where $\Delta \subseteq \Delta'$. Intuitively, $\mathcal{U}_\Delta$ contains codes of types that can vary only along dimensions in Δ. We have *universe inclusions* from $\mathcal{U}_\Delta$ to $\mathcal{U}_{\Delta'}$ whenever $\Delta \subseteq \Delta'$; in the examples we will not write these explicitly. Note that we do not have $\widehat{\mathcal{U}_\Delta} : \mathcal{U}_{\Delta'}$, i.e., these universes do not form a hierarchy. We could additionally have an orthogonal hierarchy of universes, i.e. for each clock context Δ a hierarchy of universes $\mathcal{U}^1_\Delta : \mathcal{U}^2_\Delta : \cdots$.

All judgements are closed under clock weakening and clock substitution. The former means that if, e.g., $\Gamma \vdash_\Delta t : A$ is derivable then, for any clock variable $\kappa \notin \Delta$, the judgement $\Gamma \vdash_{\Delta,\kappa} t : A$ is also derivable. The latter means that if, e.g., $\Gamma \vdash_{\Delta,\kappa} t : A$ is derivable and $\vdash_\Delta \kappa'$ then the judgement $\Gamma[\kappa'/\kappa] \vdash_\Delta t[\kappa'/\kappa] : A[\kappa'/\kappa]$ is also derivable, where clock substitution $[\kappa'/\kappa]$ is defined as obvious.

The rules for guarded recursion can be found in Figs. 3.2 and 3.3; rules for coinductive types are postponed until Sec. 3.4. Recall the 'later' type former $\triangleright$, which expresses that something will be available at a later time. In GDTT we have $\overset{\kappa}{\triangleright}$ for each clock κ, so we can delay a type along different dimensions. As discussed in the introduction, we generalise the applicative functor structure of each $\overset{\kappa}{\triangleright}$ via *delayed substitutions*, which allow a substitution to be delayed until its substituent is available. We showed in the introduction how a type with a single delayed substitution $\overset{\kappa}{\triangleright}[x \leftarrow t].A$ should work. However if we have a term f with more than one argument, for example of type $\overset{\kappa}{\triangleright}(\Pi(x : A).\Pi(y : B).C)$, and wish to type an application $f \circledast_\kappa t \circledast_\kappa u$ (where $\circledast_\kappa$ is the applicative functor operation $\circledast$ for clock κ) we may have neither t nor u available now, and so we need sequences of delayed substitutions to

$\vdash_\Delta \kappa$	valid clock	$\Gamma \vdash_\Delta A \equiv B$	type equality
$\Gamma \vdash_\Delta$	well-formed context	$\Gamma \vdash_\Delta t \equiv u : A$	term equality
$\Gamma \vdash_\Delta A\ \mathsf{type}$	well-formed type	$\vdash_\Delta \xi : \Gamma \overset{\kappa}{\to\triangleright} \Gamma'$	delayed substitution
$\Gamma \vdash_\Delta t : A$	typing judgment		

Figure 3.1: Judgements in GDTT.

define the type $\overset{\kappa}{\triangleright}[x \leftarrow t, y \leftarrow u].C$. Our concrete examples of Sec. 3.3 will show that this issue arises in practice. We therefore define sequences of delayed substitutions ξ. The new raw types, terms, and delayed substitutions of GDTT are given by the grammar

$$A, B ::= \cdots \mid \overset{\kappa}{\triangleright}\xi.A \qquad t, u ::= \cdots \mid \mathsf{next}^\kappa\, \xi.t \mid \widehat{\triangleright}\kappa t \qquad \xi ::= \cdot \mid \xi\,[x \leftarrow t].$$

Note that we just write $\overset{\kappa}{\triangleright}A$ where its delayed substitution is the empty $\cdot$, and that $\overset{\kappa}{\triangleright}\xi.A$ binds the variables substituted for by ξ in A, and similarly for next.

The three rules DS-Emp, DS-Cons, and Tf-▷ are used to construct the type $\overset{\kappa}{\triangleright}\xi.A$. These rules formulate how to generalise these types to arbitrarily long delayed substitutions. Once the type formation rule is established, the introduction rule Ty-Next is the natural one.

With delayed substitutions we can *define* $\circledast$ as

$$f \circledast t \triangleq \mathsf{next}^\kappa \begin{bmatrix} g \leftarrow f \\ x \leftarrow t \end{bmatrix}.g\,x.$$

Using the rules in Fig. 3.2 we can derive the following typing judgement for $\circledast$

$$\frac{\Gamma \vdash_\Delta f : \overset{\kappa}{\triangleright}\xi.\Pi(x : A).B \qquad \Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright}\xi.A}{\Gamma \vdash_\Delta f \circledast t : \overset{\kappa}{\triangleright}\xi[x \leftarrow t].B}\ \text{Ty-}\circledast$$

When a term has the form $\mathsf{next}^\kappa\, \xi\,[x \leftarrow \mathsf{next}^\kappa\, \xi.u].t$, then we have enough information to perform the substitution in both the term and its type. The rule TmEq-Force applies the substitution by equating the term with the result of an actual substitution, $\mathsf{next}^\kappa\, \xi.t[u/x]$. The rule TyEq-Force does the same for its type. Using TmEq-Force we can derive the basic term equality

$$(\mathsf{next}^\kappa\, \xi.f) \circledast (\mathsf{next}^\kappa\, \xi.t) \equiv \mathsf{next}^\kappa\, \xi.(f\,t).$$

typical of applicative functors [64].

It will often be the case that a delayed substitution is unnecessary, because the variable to be substituted for does not occur free in the type/term. This is what TyEq-▷-Weak and TmEq-Next-Weak express, and with these we can justify the simpler typing rule

$$\frac{\Gamma \vdash_\Delta f : \overset{\kappa}{\triangleright}\xi.(A \to B) \qquad \Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright}\xi.A}{\Gamma \vdash_\Delta f \circledast t : \overset{\kappa}{\triangleright}\xi.B}$$

In other words, delayed substitutions on the type are not necessary when we apply a non-dependent function.

Further, we have the applicative functor identity law

$$(\mathsf{next}^\kappa\, \xi.\lambda x.x) \circledast t \equiv t.$$

Universes

$$\frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \mathcal{U}_{\Delta'} \text{ type}}\ \textsc{Univ} \qquad\qquad \frac{\Gamma \vdash_\Delta A : \mathcal{U}_{\Delta'}}{\Gamma \vdash_\Delta \mathsf{El}(A) \text{ type}}\ \textsc{El}$$

Delayed substitutions:

$$\frac{\Gamma \vdash_\Delta \qquad \vdash_\Delta \kappa}{\vdash_\Delta \cdot : \Gamma \xrightarrow{\kappa}_\triangleright \cdot}\ \textsc{DS-Emp} \qquad\qquad \frac{\vdash_\Delta \xi : \Gamma \xrightarrow{\kappa}_\triangleright \Gamma' \qquad \Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright} \xi.A}{\vdash_\Delta \xi\,[x \leftarrow t] : \Gamma \xrightarrow{\kappa}_\triangleright \Gamma', x : A}\ \textsc{DS-Cons}$$

Typing rules:

$$\frac{\Gamma, \Gamma' \vdash_\Delta A \text{ type} \qquad \vdash_\Delta \xi : \Gamma \xrightarrow{\kappa}_\triangleright \Gamma'}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright} \xi.A \text{ type}}\ \textsc{Tf-}\triangleright \qquad\qquad \frac{\vdash_{\Delta'} \kappa \qquad \Gamma \vdash_\Delta A : \overset{\kappa}{\triangleright} \mathcal{U}_{\Delta'}}{\Gamma \vdash_\Delta \widehat{\triangleright \kappa} A : \mathcal{U}_{\Delta'}}\ \textsc{Ty-}\widehat{\triangleright}$$

$$\frac{\Gamma, \Gamma' \vdash_\Delta t : A \qquad \vdash_\Delta \xi : \Gamma \xrightarrow{\kappa}_\triangleright \Gamma'}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\, \xi.t : \overset{\kappa}{\triangleright} \xi.A}\ \textsc{Ty-Next} \qquad\qquad \frac{\vdash_\Delta \kappa \qquad \Gamma, x : \overset{\kappa}{\triangleright} A \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{fix}^\kappa\, x.t : A}\ \textsc{Ty-Fix}$$

Figure 3.2: Overview of the new typing rules involving $\triangleright$ and delayed substitutions.

This follows from the rule TmEq-Next-Var, which allows us to simplify a term $\mathsf{next}^\kappa\, \xi\,[y \leftarrow t].y$ to t.

Sometimes it is necessary to switch the order in the delayed substitution. Two substitutions can switch places, as long as they do not depend on each other; this is what TyEq-$\triangleright$-Exch and TmEq-Next-Exch express.

Rule TmEq-Next-Comm is not used in the examples of this paper, but it implies the rule $\mathsf{next}^\kappa\, \xi\,[x \leftarrow t].\, \mathsf{next}^\kappa\, x \equiv \mathsf{next}^\kappa\, t$, which is needed in Paviotti's PhD work.

3.2.1 Fixed points and guarded recursive types

In GDTT we have for each clock κ valid in the current clock context a fixed-point combinator fix^κ. This differs from a traditional fixed-point combinator in that the type of the recursion variable is not the same as the result type; instead its type is *guarded* with $\overset{\kappa}{\triangleright}$. When we define a term using the fixed-point, we say that it is defined by *guarded recursion*. When the term is intuitively a proof, we say we are proving by *Löb induction* [7].

Guarded recursive types are defined as fixed-points of suitably guarded functions on universes. This is the approach of Birkedal and Møgelberg [10], but the generality of the rules of GDTT allows us to define more interesting dependent guarded recursive types, for example the predicates of Sec. 3.3.

Definitional type equalities:

$$\overset{\kappa}{\triangleright}\xi\,[x \leftarrow t].A \equiv \overset{\kappa}{\triangleright}\xi.A \qquad \text{(TyEq-▷-Weak)}$$

$$\overset{\kappa}{\triangleright}\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A \equiv \overset{\kappa}{\triangleright}\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A \qquad \text{(TyEq-▷-Exch)}$$

$$\overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t].A \equiv \overset{\kappa}{\triangleright}\xi.A[t/x] \qquad \text{(TyEq-Force)}$$

$$\mathsf{El}(\widehat{\triangleright}\kappa\,(\mathsf{next}^\kappa\,\xi.t)) \equiv \overset{\kappa}{\triangleright}\xi.\,\mathsf{El}(t) \qquad \text{(TyEq-El-▷)}$$

$$\mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}\,(\mathsf{next}^\kappa\,\xi.t, \mathsf{next}^\kappa\,\xi.s) \equiv \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A\,(t,s) \qquad \text{(TyEq-▷)}$$

Definitional term equalities:

$$\mathsf{next}^\kappa\,\xi\,[x \leftarrow t].u \equiv \mathsf{next}^\kappa\,\xi.u \qquad \text{(TmEq-Next-Weak)}$$

$$\mathsf{next}^\kappa\,\xi\,[x \leftarrow t].x \equiv t \qquad \text{(TmEq-Next-Var)}$$

$$\mathsf{next}^\kappa\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}^\kappa\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v \qquad \text{(TmEq-Next-Exch)}$$

$$\mathsf{next}^\kappa\,\xi.\mathsf{next}^\kappa\,\xi'.u \equiv \mathsf{next}^\kappa\,\xi'.\mathsf{next}^\kappa\,\xi.u \qquad \text{(TmEq-Next-Comm)}$$

$$\mathsf{next}^\kappa\,\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t].u \equiv \mathsf{next}^\kappa\,\xi.u[t/x] \qquad \text{(TmEq-Force)}$$

$$\mathsf{fix}^\kappa\,x.t \equiv t[\mathsf{next}^\kappa\,(\mathsf{fix}^\kappa\,x.t)/x] \qquad \text{(TmEq-Fix)}$$

Figure 3.3: New type and term equalities in GDTT. Rules TyEq-▷-Weak and TmEq-Next-Weak require that A and u are well-formed in a context without x. Rules TyEq-▷-Exch and TmEq-Next-Exch assume that exchanging x and y is allowed, i.e., that the type of x does not depend on y and vice versa. Likewise, rule TmEq-Next-Comm assumes that exchanging the codomains of ξ and ξ' is allowed and that none of the variables in the codomains of ξ and ξ' appear in the type of u.

We first illustrate the technique by defining the (non-dependent) type of guarded streams. Recall from the introduction that we want the type of guarded streams, for clock κ, to satisfy the equation $\mathsf{Str}^\kappa_A \equiv A \times \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A$.

The type A will be equal to $\mathsf{El}(B)$ for some code B in some universe $\mathcal{U}_\Delta$ where the clock variable κ is not in Δ. We then define the *code* S^κ_A of Str^κ_A in the universe $\mathcal{U}_{\Delta,\kappa}$ to be $S^\kappa_A \triangleq \mathsf{fix}^\kappa\,X.B\widehat{\times}\widehat{\triangleright}\kappa X$, where $\widehat{\times}$ is the code of the (simple) product type. Via the rules of GDTT we can show $\mathsf{Str}^\kappa_A \simeq A \times \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A$ as desired.

The head and tail operations, $\mathsf{hd}^\kappa : \mathsf{Str}^\kappa_A \to A$ and $\mathsf{tl}^\kappa : \mathsf{Str}^\kappa_A \to \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A$ are simply the first and the second projections. Conversely, we construct streams by pairing. We use the suggestive cons^κ notation which we define as

$$\mathsf{cons}^\kappa : A \to \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A \to \mathsf{Str}^\kappa_A \qquad \mathsf{cons}^\kappa \triangleq \lambda\,(a : A)\left(as : \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A\right).\langle a, as\rangle$$

Defining guarded streams is also done via guarded recursion, for example the stream consisting only of ones is defined as $\mathsf{ones} \triangleq \mathsf{fix}^\kappa\,x.\mathsf{cons}^\kappa\,1\,x$.

The rule TyEq-El-▷ is essential for defining guarded recursive types as fixed-points on universes, and it can also be used for defining more advanced guarded recursive dependent types such as covectors; see Sec. 3.3.

3.2.2 Identity types

GDTT has standard extensional identity types $\mathsf{Id}_A(t,u)$ (see, e.g., Jacobs [47]) but with two additional type equivalences necessary for working with guarded dependent types. We write $\mathsf{r}_A\, t$ for the reflexivity proof $\mathsf{Id}_A(t,t)$. The first type equivalence is the rule TyEq-▷. This rule, which is validated by the model of Sec. 3.6, may be thought of by analogy to type equivalences often considered in homotopy type theory [84], such as

$$\mathsf{Id}_{A\times B}(\langle s_1,s_2\rangle,\langle t_1,t_2\rangle) \equiv \mathsf{Id}_A(s_1,t_1)\times \mathsf{Id}_B(s_2,t_2). \tag{3.2}$$

There are two important differences. The first is that (3.2) is (using univalence) a propositional type equality, whereas TyEq-▷ specifices a definitional type equality. This is natural in an extensional type theory. The second difference is that there are terms going in both directions in (3.2), whereas we would have a term of type $\mathsf{Id}_{\blacktriangleright^\kappa \xi.A}(\mathsf{next}^\kappa\,\xi.t, \mathsf{next}^\kappa\,\xi.u) \to \blacktriangleright^\kappa \xi.\mathsf{Id}_A(t,u)$ without the rule TyEq-▷.

The second novel type equality rule, which involves clock quantification, will be presented in Sec. 3.4.

3.3 Examples

In this section we present some example terms typable in GDTT. Our examples will use a term, which we call $\mathsf{p}\eta$, of type $\Pi(s,t : A\times B).\mathsf{Id}_A(\pi_1 t,\pi_1 s) \to \mathsf{Id}_B(\pi_2 t,\pi_2 s) \to \mathsf{Id}_{A\times B}(t,s)$. This term is definable in any type theory with a strong (dependent) elimination rule for dependent sums. The second property we will use is that $\mathsf{Str}^\kappa_A \equiv A \times \blacktriangleright^\kappa \mathsf{Str}^\kappa_A$. Because hd^κ and tl^κ are simply first and second projections, $\mathsf{p}\eta$ also has type $\Pi\left(xs,ys : \mathsf{Str}^\kappa_A\right).\mathsf{Id}_A(\mathsf{hd}^\kappa\, xs, \mathsf{hd}^\kappa\, ys) \to \mathsf{Id}_{\blacktriangleright^\kappa \mathsf{Str}^\kappa_A}(\mathsf{tl}^\kappa\, xs, \mathsf{tl}^\kappa\, ys) \to \mathsf{Id}_{\mathsf{Str}^\kappa_A}(xs,ys)$.

$\mathsf{zipWith}^\kappa$ preserves commutativity. In GDTT we define the $\mathsf{zipWith}^\kappa$ function which has the type $(A\to B\to C)\to \mathsf{Str}^\kappa_A \to \mathsf{Str}^\kappa_B \to \mathsf{Str}^\kappa_C$ by

$$\mathsf{zipWith}^\kappa\, f \triangleq \mathsf{fix}^\kappa\, \varphi.\lambda xs,ys.\,\mathsf{cons}^\kappa\,(f\,(\mathsf{hd}^\kappa\, xs)\,(\mathsf{hd}^\kappa\, ys))\,(\varphi \circledast \mathsf{tl}^\kappa\, xs \circledast \mathsf{tl}^\kappa\, ys).$$

We show that commutativity of f implies commutativity of $\mathsf{zipWith}^\kappa\, f$, i.e., that

$$\Pi(f : A\to A\to B).\,(\Pi(x,y:A).\mathsf{Id}_B(f\,x\,y, f\,y\,x)) \to$$
$$\Pi\left(xs,ys : \mathsf{Str}^\kappa_A\right).\mathsf{Id}_{\mathsf{Str}^\kappa_B}(\mathsf{zipWith}^\kappa\, f\, xs\, ys, \mathsf{zipWith}^\kappa\, f\, ys\, xs)$$

is inhabited. The term that inhabits this type is

$$\lambda f.\lambda c.\,\mathsf{fix}^\kappa\,\varphi.\lambda xs, ys.\,\mathsf{p}\eta\,(c\,(\mathsf{hd}^\kappa\,xs)\,(\mathsf{hd}^\kappa\,ys))\,(\varphi\circledast\mathsf{tl}^\kappa\,xs\circledast\mathsf{tl}^\kappa\,ys).$$

Here, φ has type $\overset{\kappa}{\triangleright}(\Pi\big(xs, ys : \mathsf{Str}_A^\kappa\big).\mathsf{Id}_{\mathsf{Str}_B^\kappa}(\mathsf{zipWith}^\kappa\,f\,xs\,ys, \mathsf{zipWith}^\kappa\,f\,ys\,xs))$ so to type the term above, we crucially need delayed substitutions.

An example with covectors. The next example is more sophisticated, as it involves programming and proving with a data type that, unlike streams, is dependently typed. Indeed the generalised later, carrying a delayed substitution, is necessary to type even elementary programs. *Covectors* are the potentially infinite version of vectors (lists with length). To define guarded covectors we first need guarded co-natural numbers. The definition in GDTT is $\mathsf{CoIN}^\kappa \triangleq \mathsf{El}\big(\mathsf{fix}^\kappa\,X.(\widehat{\mathbf{1}+\triangleright}\kappa X)\big)$; this type satisfies $\mathsf{CoIN}^\kappa \equiv \mathbf{1} + \overset{\kappa}{\triangleright}\mathsf{CoIN}^\kappa$. Using CoIN^κ we can define the type family of covectors $\mathsf{CoVec}_A^\kappa\,n \triangleq \mathsf{El}(\widehat{\mathsf{CoVec}}_A^\kappa\,n)$, where

$$\begin{aligned}\widehat{\mathsf{CoVec}}_A^\kappa \triangleq\, &\mathsf{fix}^\kappa\Big(\varphi : \overset{\kappa}{\triangleright}(\mathsf{CoIN}^\kappa \to \mathcal{U}_{\Delta,\kappa})\Big).\lambda(n : \mathsf{CoIN}^\kappa).\,\mathsf{case}\,n\,\mathsf{of}\\ &\mathsf{inl}\,u \Rightarrow \widehat{\mathbf{1}}\\ &\mathsf{inr}\,m \Rightarrow A\,\widehat{\times\triangleright}\,\kappa(\varphi\circledast m).\end{aligned}$$

We will not distinguish between CoVec_A^κ and $\widehat{\mathsf{CoVec}}_A^\kappa$. As an example of covectors, we define ones of type $\Pi(n : \mathsf{CoIN}^\kappa).\,\mathsf{CoVec}_{\mathbb{N}}^\kappa\,n$ which produces a covector of any length consisting only of ones:

$$\mathsf{ones} \triangleq \mathsf{fix}^\kappa\,\varphi.\lambda(n : \mathsf{CoIN}^\kappa).\,\mathsf{case}\,n\,\mathsf{of}\,\{\mathsf{inl}\,u \Rightarrow \mathsf{inl}\,\langle\rangle; \mathsf{inr}\,m \Rightarrow \langle 1, \varphi\circledast m\rangle\}.$$

Although this is one of the simplest covector programs one can imagine, it does not type-check without the generalised later with delayed substitutions.

The map function on covectors is defined as

$$\begin{aligned}\mathsf{map}\;&:(A\to B)\to\Pi(n:\mathsf{CoIN}^\kappa).\,\mathsf{CoVec}_A^\kappa\,n\to\mathsf{CoVec}_B^\kappa\,n\\ \mathsf{map}\,f\;&\triangleq\mathsf{fix}^\kappa\,\varphi.\lambda(n:\mathsf{CoIN}^\kappa).\,\mathsf{case}\,n\,\mathsf{of}\\ &\quad\mathsf{inl}\,u\Rightarrow\lambda(x:1).x\\ &\quad\mathsf{inr}\,m\Rightarrow\lambda\Big(p:A\times\overset{\kappa}{\triangleright}[n\leftarrow m].(\mathsf{CoVec}_A^\kappa\,n)\Big).\langle f\,(\pi_1 p),\varphi\circledast m\circledast(\pi_2 p)\rangle.\end{aligned}$$

It preserves composition: the following type is inhabited

$$\begin{aligned}&\Pi(f:A\to B)(g:B\to C)(n:\mathsf{CoIN}^\kappa)(xs:\mathsf{CoVec}_A^\kappa\,n).\\ &\qquad\mathsf{Id}_{\mathsf{CoVec}_C^\kappa\,n}(\mathsf{map}\,g\,n\,(\mathsf{map}\,f\,n\,xs),\mathsf{map}\,(g\circ f)\,n\,xs)\end{aligned}$$

by the term

$$\begin{aligned}&\lambda(f:A\to B)(g:B\to C).\,\mathsf{fix}^\kappa\,\varphi.\lambda(n:\mathsf{CoIN}^\kappa).\,\mathsf{case}\,n\,\mathsf{of}\\ &\;\mathsf{inl}\,u\Rightarrow\lambda(xs:1).\mathsf{r}_1\,xs\\ &\;\mathsf{inr}\,m\Rightarrow\lambda(xs:\mathsf{CoVec}_A^\kappa(\mathsf{inr}\,m)).\,\mathsf{p}\eta\,(\mathsf{r}_C\,g(f\,(\pi_1 xs)))\,(\varphi\circledast m\circledast\pi_2 xs).\end{aligned}$$

3.4 Coinductive types

As discussed in the introduction, guarded recursive types on their own disallow productive but acausal function definitions. To capture such functions we need to be able to remove $\overset{\kappa}{\triangleright}$. However such eliminations must be controlled to avoid trivialising $\overset{\kappa}{\triangleright}$. If we had an unrestricted elimination term $\mathsf{elim} : \overset{\kappa}{\triangleright} A \to A$ every type would be inhabited via fix^κ, making the type theory inconsistent.

However, we may eliminate $\overset{\kappa}{\triangleright}$ provided that the term does not depend on the clock κ, i.e., the term is typeable in a context where κ does not appear. Intuitively, such contexts have no temporal properties along the κ dimension, so we may progress the computation without violating guardedness. Fig. 3.4 extends the system of Fig. 3.2 to allow the removal of clocks in such a setting, by introducing *clock quantifiers* $\forall\kappa$ [8, 19, 66]. This is a binding construct with associated term constructor $\Lambda\kappa$, which also binds κ. The elimination term is *clock application*. Application of the term t of type $\forall\kappa.A$ to a clock κ is written as $t[\kappa]$. One may think of $\forall\kappa.A$ as analogous to the type $\forall\alpha.A$ in polymorphic lambda calculus; indeed the basic rules are precisely the same, but we have an additional construct $\mathsf{prev}\,\kappa.t$, called 'previous', to allow removal of the later modality $\overset{\kappa}{\triangleright}$.

Typing this new construct $\mathsf{prev}\,\kappa.t$ is somewhat complicated, as it requires 'advancing' a delayed substitution, which turns it into a context morphism (an actual substitution); see Fig. 3.5 for the definition. The judgement $\rho :_\Delta \Gamma \to \Gamma'$ expresses that ρ is a context morphism from context $\Gamma \vdash_\Delta$ to the context $\Gamma' \vdash_\Delta$. We use the notation $\rho[t/x]$ for extending the context morphism by mapping the variable x to the term t. We illustrate this with two concrete examples.

First, we can indeed remove later under a clock quantifier:

$$\mathsf{force} : \forall\kappa.\overset{\kappa}{\triangleright} A \to \forall\kappa.A \qquad\qquad \mathsf{force} \triangleq \lambda x.\,\mathsf{prev}\,\kappa.x[\kappa].$$

The type is correct because advancing the empty delayed substitution in $\overset{\kappa}{\triangleright}$ turns it into the identity substitution ι, and $A\iota \equiv A$. The β and η rules ensure that force is the inverse to the canonical term $\lambda x.\Lambda\kappa.\,\mathsf{next}^\kappa\, x[\kappa]$ of type $\forall\kappa.A \to \forall\kappa.\overset{\kappa}{\triangleright} A$.

Second, we may see an example with a non-empty delayed substitution in the term $\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,\lambda n.\,\mathsf{succ}\,n \circledast \mathsf{next}^\kappa\,0$ of type $\forall\kappa.\mathbb{N}$. Recall that $\circledast$ is syntactic sugar and so more precisely the term is

$$\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa \left[\begin{array}{l} f \leftarrow \mathsf{next}^\kappa\,\lambda n.\,\mathsf{succ}\,n \\ x \leftarrow \mathsf{next}^\kappa\,0 \end{array}\right].f\,x. \tag{3.3}$$

Advancing the delayed substitution turns it into the substitution mapping the variable f to the term $(\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,\lambda n.\,\mathsf{succ}\,n)[\kappa]$ and the variable x to the term $(\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,0)[\kappa]$. Using the β rule for prev, then the β rule for $\forall\kappa$, this simplifies to the substitution mapping f to $\lambda n.\,\mathsf{succ}\,n$ and x to 0. With this we

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} A\ \mathsf{type}}{\Gamma \vdash_\Delta \forall\kappa.A\ \mathsf{type}}\ \textsc{Ty-}\forall \qquad \frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash_\Delta t : \forall\kappa.\mathcal{U}_{\Delta',\kappa}}{\Gamma \vdash_\Delta \widehat{\forall}\, t : \mathcal{U}_{\Delta'}}\ \textsc{Ty-}\forall\textsc{-code}$$

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_\Delta \Lambda\kappa.t : \forall\kappa.A}\ \textsc{Ty-}\Lambda \qquad \frac{\vdash_\Delta \kappa' \qquad \Gamma \vdash_\Delta t : \forall\kappa.A}{\Gamma \vdash_\Delta t[\kappa'] : A[\kappa'/\kappa]}\ \textsc{Ty-app}$$

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : {\overset{\kappa}{\triangleright}}\xi.A}{\Gamma \vdash_\Delta \mathsf{prev}\,\kappa.t : \forall\kappa.(A(\mathsf{adv}^\kappa_\Delta(\xi)))}\ \textsc{Ty-prev}$$

Figure 3.4: Overview of the new typing rules for coinductive types.

have that the term (3.3) is equal to $\Lambda\kappa.((\lambda n.\,\mathsf{succ}\,n)\,0)$ which is in turn equal to $\Lambda\kappa.1$.

An important property of the term $\mathsf{prev}\,\kappa.t$ is that κ is *bound* in t; hence $\mathsf{prev}\,\kappa.t$ has type $\forall\kappa.A$ instead of just A. This ensures that substitution of terms in types and terms is well-behaved and we do not need the explicit substitutions used, for example, by Clouston et al. [27] where the unary type-former $\Box$ was used in place of clocks. This binding structure ensures, for instance, that the introduction rule Ty-Λ closed under substitution in Γ.

The rule TmEq-$\forall$-fresh states that if t has type $\forall\kappa.A$ and the clock κ does not appear in the *type* A, then it does not matter to which clock t is applied, as the resulting term will be the same. In the polymorphic lambda calculus, the corresponding rule for universal quantification over types would be a consequence of relational parametricity.

We further have the construct $\widehat{\forall}$ and the rule Ty-$\forall$-code which witness that the universes are closed under $\forall\kappa$.

To summarise, the new raw types and terms, extending those of Sec. 3.2, are

$$A, B ::= \cdots \mid \forall\kappa.A \qquad\qquad t, u ::= \cdots \mid \Lambda\kappa.t \mid t[\kappa] \mid \widehat{\forall}\, t \mid \mathsf{prev}\,\kappa.t$$

Finally, we have the equality rule TyEq-$\forall$-Id analogous to the rule TyEq-$\triangleright$. Note that, as in Sec. 3.2.2, there is a canonical term of type $\mathsf{Id}_{\forall\kappa.A}(t,s) \to \forall\kappa.\mathsf{Id}_A(t[\kappa], s[\kappa])$ but, without this rule, no term in the reverse direction.

3.4.1 Derivable type isomorphisms

The encoding of coinductive types using guarded recursive types crucially uses a family of type isomorphisms commuting $\forall\kappa$ over other type formers [8, 66]. By a type isomorphism $A \cong B$ we mean two well-typed terms f and g of types $f : A \to B$ and $g : B \to A$ such that $f(g\,x) \equiv x$ and $g(f\,x) \equiv x$. The

$$\frac{\vdash_{\Delta,\kappa} \cdot : \Gamma \overset{\kappa}{\to} \cdot \qquad \Gamma \vdash_\Delta}{\mathsf{adv}^\kappa_\Delta(\cdot) \triangleq \iota :_{\Delta,\kappa} \Gamma \to \Gamma}$$

$$\frac{\vdash_{\Delta,\kappa} \xi[x \leftarrow t] : \Gamma \overset{\kappa}{\to} \Gamma', x : A \qquad \Gamma \vdash_\Delta}{\mathsf{adv}^\kappa_\Delta(\xi[x \leftarrow t]) \triangleq \mathsf{adv}^\kappa_\Delta(\xi)[(\mathsf{prev}\,\kappa.t)[\kappa]/x] :_{\Delta,\kappa} \Gamma \to \Gamma, \Gamma', x : A}$$

Figure 3.5: Advancing a delayed substitution.

first type isomorphism is $\forall\kappa.A \cong A$ whenever κ is not free in A. The terms $g = \lambda x.\Lambda\kappa.x$ of type $A \to \forall\kappa.A$ and $f = \lambda x.x[\kappa_0]$ of type $A \to \forall\kappa.A$ witness the isomorphism. Note that we used the clock constant κ_0 in an essential way. The equality $f\,(g\,x) \equiv x$ follows using only the β rule for clock application. The equality $g(f\,x) \equiv x$ follows using by the rule TmEq-$\forall$-fresh.

The following type isomorphisms follow by using β and η laws for the constructs involved.

- If $\kappa \notin A$ then $\forall\kappa.\Pi(x : A).B \cong \Pi(x : A).\forall\kappa.B$.
- $\forall\kappa.\Sigma\,(x : A)\,B \cong \Sigma\,(y : \forall\kappa.A)\,(\forall\kappa.B\,[y[\kappa]/x])$.
- $\forall\kappa.A \cong \forall\kappa.{}^{\kappa}\!\triangleright A$.

There is an important additional type isomorphism witnessing that $\forall\kappa$ commutes with binary sums; however unlike the isomorphisms above we require equality reflection to show that the two functions are inverse to each other up to definitional equality. There is a canonical term of type $\forall\kappa.A + \forall\kappa.B \to \forall\kappa.(A + B)$ using just ordinary elimination of coproducts. Using the fact that we encode binary coproducts using Σ-types and universes we can define a term com^+ of type $\forall\kappa.(A + B) \to \forall\kappa.A + \forall\kappa.B$ which is a inverse to the canonical term. In particular com^+ satisfies the following two equalities which will be used below.

$$\mathsf{com}^+\,(\Lambda\kappa.\,\mathsf{inl}\,t) \equiv \mathsf{inl}\,\Lambda\kappa.t \qquad \mathsf{com}^+\,(\Lambda\kappa.\,\mathsf{inr}\,t) \equiv \mathsf{inr}\,\Lambda\kappa.t. \tag{3.4}$$

3.5 Example programs with coinductive types

Let A be a type with code $\widehat{A}$ in clock context Δ and κ a fresh clock variable. Let $\mathsf{Str}_A = \forall\kappa.\,\mathsf{Str}^\kappa_A$. We can define head, tail and cons functions

$$\begin{aligned}
\mathsf{hd} &: \mathsf{Str}_A \to A & \mathsf{hd} &\triangleq \lambda xs.\,\mathsf{hd}^{\kappa_0}\,(xs[\kappa_0]) \\
\mathsf{tl} &: \mathsf{Str}_A \to \mathsf{Str}_A & \mathsf{tl} &\triangleq \lambda xs.\,\mathsf{prev}\,\kappa.\,\mathsf{tl}^\kappa\,(xs[\kappa]) \\
\mathsf{cons} &: A \to \mathsf{Str}_A \to \mathsf{Str}_A & \mathsf{cons} &\triangleq \lambda x.\lambda xs.\Lambda\kappa.\,\mathsf{cons}^\kappa\,x\,(\mathsf{next}^\kappa\,(xs[\kappa])).
\end{aligned}$$

Definitional type equalities:

$$\frac{\Gamma \vdash_\Delta \qquad \Delta' \subseteq \Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : \mathcal{U}_{\Delta',\kappa}}{\Gamma \vdash_\Delta \mathsf{El}(\widehat{\forall}\,\Lambda\kappa.t) \equiv \forall\kappa.\mathsf{El}(t)}\ \textsc{TyEq-}\forall\textsc{-el}$$

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} A\ \mathsf{type} \qquad \Gamma \vdash_\Delta t : \forall\kappa.A \qquad \Gamma \vdash_\Delta s : \forall\kappa.A}{\Gamma \vdash_\Delta \forall\kappa.\mathsf{Id}_A(t[\kappa], s[\kappa]) \equiv \mathsf{Id}_{\forall\kappa.A}(t,s)}\ \textsc{TyEq-}\forall\textsc{-Id}$$

Definitional term equalities:

$$\frac{\Gamma \vdash_\Delta \qquad \vdash_\Delta \kappa' \qquad \Gamma \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_\Delta (\Lambda\kappa.t)[\kappa'] \equiv t[\kappa'/\kappa] : A[\kappa'/\kappa]}\ \textsc{TmEq-}\forall\textsc{-}\beta$$

$$\frac{\kappa \notin \Delta \qquad \Gamma \vdash_\Delta t : \forall\kappa.A}{\Gamma \vdash_\Delta \Lambda\kappa.t[\kappa] \equiv t : \forall\kappa.A}\ \textsc{TmEq-}\forall\textsc{-}\eta$$

$$\frac{\kappa \notin \Delta \qquad \Gamma \vdash_\Delta A\ \mathsf{type} \qquad \Gamma \vdash_\Delta t : \forall\kappa.A \qquad \vdash_\Delta \kappa' \qquad \vdash_\Delta \kappa''}{\Gamma \vdash_\Delta t[\kappa'] \equiv t[\kappa''] : A}\ \textsc{TmEq-}\forall\textsc{-fresh}$$

$$\frac{\Gamma \vdash_\Delta \qquad \vdash_{\Delta,\kappa} \xi : \Gamma \overset{\kappa}{\to} \Gamma' \qquad \Gamma, \Gamma' \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_\Delta \mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,\xi.t \equiv \Lambda\kappa.t(\mathsf{adv}^\kappa_\Delta(\xi)) : \forall\kappa.(A(\mathsf{adv}^\kappa_\Delta(\xi)))}\ \textsc{TmEq-prev-}\beta$$

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : \overset{\kappa}{\triangleright} A}{\Gamma \vdash_{\Delta,\kappa} \mathsf{next}^\kappa((\mathsf{prev}\,\kappa.t)[\kappa]) \equiv t : \overset{\kappa}{\triangleright} A}\ \textsc{TmEq-prev-}\eta$$

Figure 3.6: Type and term equalities involving clock quantification.

With these we can define the *acausal* 'every other' function eo^κ that removes every second element of the input stream. It is acausal because the second element of the output stream is the third element of the input. Therefore to type the function we need to have the input stream always available, so clock quantification must be used. The function eo^κ of type $\mathsf{Str}_A \to \mathsf{Str}^\kappa_A$ is defined as

$$\mathsf{eo}^\kappa \triangleq \mathsf{fix}^\kappa\,\varphi.\lambda(xs : \mathsf{Str}_A).\,\mathsf{cons}^\kappa(\mathsf{hd}\,xs)(\varphi \circledast \mathsf{next}^\kappa((\mathsf{tl}(\mathsf{tl}\,xs)))).$$

The result is a *guarded* stream, but we can easily strengthen it and define eo of type $\mathsf{Str}_A \to \mathsf{Str}_A$ as $\mathsf{eo} \triangleq \lambda xs.\Lambda\kappa.\,\mathsf{eo}^\kappa\,xs$.

We can also work with covectors (not just guarded covectors as in Sec. 3.3). This is a dependent coinductive type indexed by conatural numbers which is the type $\mathsf{CoIN} = \forall\kappa.\,\mathsf{CoIN}^\kappa$. It is easy to define $\overline{0}$ and $\overline{\mathsf{succ}}$ as $\overline{0} \triangleq \Lambda\kappa.\,\mathsf{inl}\,\langle\rangle$ and $\overline{\mathsf{succ}} \triangleq \lambda n.\Lambda\kappa.\,\mathsf{inr}\,(\mathsf{next}^\kappa\,(n[\kappa]))$. Next, we can define a transport function

$\mathsf{com}^{\mathsf{CoIN}}$ of type $\mathsf{com}^{\mathsf{CoIN}} : \mathsf{CoIN} \to 1 + \mathsf{CoIN}$ satisfying

$$\mathsf{com}^{\mathsf{CoIN}}\,\overline{0} \equiv \mathsf{inl}\,\langle\rangle \qquad\qquad \mathsf{com}^{\mathsf{CoIN}}(\overline{\mathsf{succ}}\,n) \equiv \mathsf{inr}\,n. \tag{3.5}$$

This function is used to define the type family of covectors as $\mathsf{CoVec}_A\,n \triangleq \forall\kappa.\,\mathsf{CoVec}_A^\kappa\,n$ where $\mathsf{CoVec}_A^\kappa : \mathsf{CoIN} \to \mathcal{U}_{\Delta,\kappa}$ is the term

$$\mathsf{fix}^\kappa\,\varphi.\lambda\,(n : \mathsf{CoIN}).\,\mathsf{case}\,\mathsf{com}^{\mathsf{CoIN}}\,n\,\mathsf{of}\left\{\mathsf{inl}\_ \Rightarrow \widehat{1}; \mathsf{inr}\,n \Rightarrow A\widehat{\times\triangleright}\kappa\,(\varphi\circledast(\mathsf{next}^\kappa\,n))\right\}.$$

Using term equalities (3.4) and (3.5) we can derive the type isomorphisms

$$\begin{aligned}\mathsf{CoVec}_A\,\overline{0} &\equiv \forall\kappa.1 \cong 1\\ \mathsf{CoVec}_A\,(\overline{\mathsf{succ}}\,n) &\equiv \forall\kappa.\left(A \times {\overset{\kappa}{\triangleright}}\left(\mathsf{CoVec}_A^\kappa\,n\right)\right) \cong A \times \mathsf{CoVec}_A\,n\end{aligned} \tag{3.6}$$

which are the expected properties of the type of covectors.

A simple function we can define is the tail function

$$\mathsf{tl} : \mathsf{CoVec}_A(\overline{\mathsf{succ}}\,n) \to \mathsf{CoVec}_A \qquad\qquad \mathsf{tl} \triangleq \lambda v.\,\mathsf{prev}\,\kappa.\pi_2\,(v[\kappa]).$$

Note that (3.6) is needed to type tl. The map function of type

$$\mathsf{map} : (A \to B) \to \Pi(n : \mathsf{CoIN}).\,\mathsf{CoVec}_A\,n \to \mathsf{CoVec}_B\,n$$

is defined as $\mathsf{map}\,f \triangleq \lambda n.\lambda xs.\Lambda\kappa.\,\mathsf{map}^\kappa\,f\,n\,(xs[\kappa])$ where map^κ is

$$\begin{aligned}&\mathsf{map}^\kappa : (A \to B) \to \Pi(n : \mathsf{CoIN}).\,\mathsf{CoVec}_A^\kappa\,n \to \mathsf{CoVec}_B^\kappa\,n\\ &\mathsf{map}^\kappa = \lambda f.\,\mathsf{fix}^\kappa\,\varphi.\lambda n.\,\mathsf{case}\,\mathsf{com}^{\mathsf{CoIN}}\,n\,\mathsf{of}\\ &\qquad\qquad\qquad\qquad \mathsf{inl}\_ \Rightarrow \lambda v.v\\ &\qquad\qquad\qquad\qquad \mathsf{inr}\,n \Rightarrow \lambda v.\,\langle f\,(\pi_1 v), \varphi\circledast(\mathsf{next}^\kappa\,n)\circledast\pi_2(v)\rangle.\end{aligned}$$

3.5.1 Lifting guarded functions

In this section we show how in general we may lift a function on guarded recursive types, such as addition of guarded streams, to a function on coinductive streams. Moreover, we show how to lift proofs of properties, such as the commutativity of addition, from guarded recursive types to coinductive types.

Let Γ be a context in clock context Δ and κ a fresh clock. Suppose A and B are types such that $\Gamma \vdash_{\Delta,\kappa} A$ type and $\Gamma, x : A \vdash_{\Delta,\kappa} B$ type. Finally let f be a function of type $\Gamma \vdash_{\Delta,\kappa} f : \Pi(x : A).B$. We define $\mathfrak{L}(f)$ satisfying the typing judgement $\Gamma \vdash_\Delta \mathfrak{L}(f) : \Pi(y : \forall\kappa.A).\forall\kappa.\,(B\,[y[\kappa]/x])$ as $\mathfrak{L}(f) \triangleq \lambda y.\Lambda\kappa.f\,(y[\kappa])$.

Now assume that f' is another term of type $\Pi(x : A).B$ (in the same context) and that we have proved $\Gamma \vdash_{\Delta,\kappa} p : \Pi(x : A).\mathsf{Id}_B\,(f\,x, f'\,x)$. As above we can give the term $\mathfrak{L}(p)$ the type $\Pi(y : \forall\kappa.A).\forall\kappa.\mathsf{Id}_{B[y[\kappa]/x]}\,(f\,(y[\kappa]), f'(y[\kappa]))$. which by using the type equality TyEq-$\forall$-Id and the η rule for $\forall$ is equal to the type

$\Pi(y : \forall\kappa.A).\mathsf{Id}_{\forall\kappa.B[y[\kappa]/x]}(\mathfrak{L}(f)\,y, \mathfrak{L}(f')\,y)$. So we have derived a property of lifted functions $\mathfrak{L}(f)$ and $\mathfrak{L}(f')$ from the properties of the guarded versions f and f'. This is a standard pattern. Using Löb induction we prove a property of a function whose result is a "guarded" type and derive the property for the lifted function.

For example we can lift the zipWith function from guarded streams to coinductive streams and prove that it preserves commutativity, using the result on guarded streams of Sec. 3.3.

3.6 Soundness

GDTT can be shown to be sound with respect to a denotational model interpreting the type theory. The model is a refinement of Bizjak and Møgelberg's [19] but for reasons of space we leave the description of a full model of GDTT for future work. Instead, to provide some intuition for the semantics of delayed substitutions, we just describe how to interpret the rule

$$\frac{x : A \vdash B\ \mathsf{type} \qquad \vdash t : \triangleright A}{\vdash \triangleright[x \leftarrow t].B\ \mathsf{type}} \tag{3.7}$$

in the case where we only have one clock available.

The subsystem of GDTT with only one clock can be modelled in the category $\mathcal{S}$, known as the topos of trees [14], the presheaf category over the first infinite ordinal ω. The objects X of $\mathcal{S}$ are families of sets $X_1, X_2, \ldots$ indexed by the positive integers, together with families of *restriction functions* $r_i^X : X_{i+1} \to X_i$ indexed similarly. There is a functor $\blacktriangleright : \mathcal{S} \to \mathcal{S}$ which maps an object X to the object

$$1 \xleftarrow{!} X_1 \xleftarrow{r_1^X} X_2 \xleftarrow{r_2^X} X_3 \longleftarrow \cdots$$

where ! is the unique map into the terminal object.

In this model, a closed type A is interpreted as an object of $\mathcal{S}$ and the type $x : A \vdash B\ \mathsf{type}$ is interpreted as an indexed family of sets $B_i(a)$, for a in A_i together with maps $r_i^B(a) \colon B_{i+1}(a) \to B_i(r_i^A(a))$. The term t in (3.7) is interpreted as a morphism $t : 1 \to \triangleright A$ so $t_i(*)$ is an element of A_i (here we write $*$ for the element of 1).

The type $\vdash \triangleright[x \leftarrow t].B\ \mathsf{type}$ is then interpreted as the object X, defined by

$$X_1 = 1 \qquad\qquad X_{i+1} = B_i(t_{i+1}(*)).$$

Notice that the delayed substitution is interpreted by substitution (reindexing) in the model; the change of the index in the model (B_i is reindexed along $t_{i+1}(*)$) corresponds to the delayed substitution in the type theory. Further notice that if B does not depend on x, then the interpretation of $\vdash \triangleright[x \leftarrow t].B\ \mathsf{type}$

reduces to the interpretation $\triangleright B$, which is defined to be $\blacktriangleright$ applied to the interpretation of B.

The above can be generalised to work for general contexts and sequences of delayed substitutions, and one can then validate that the definitional equality rules do indeed hold in this model.

3.7 Related Work

Birkedal et al. [14] introduced dependent type theory with the $\triangleright$ modality, with semantics in the topos of trees. The guardedness requirement was expressed using the syntactic check that every occurrence of a type variable lies beneath a $\triangleright$. This requirement was subsequently refined by Birkedal and Møgelberg [10], who showed that guarded recursive types could be constructed via fixed-points of functions on universes. However, the rules considered in these papers do not allow one to apply terms of type $\triangleright(\Pi(x : A).B)$, as the applicative functor construction $\circledast$ was defined only for simple function spaces. They are therefore less expressive for both programming (consider the covector ones, and function map, of Sec. 3.3) and proving, noting the extensive use of delayed substitutions in our example proofs. They further do not consider coinductive types, and so are restricted to causal functions.

The extension to coinductive types, and hence acausal functions, is due to Atkey and McBride [8], who introduced *clock quantifiers* into a simply typed setting with guarded recursion. Møgelberg [66] extended this work to dependent types and Bizjak and Møgelberg [19] refined the model further to allow clock synchronisation.

Clouston et al. [27] introduced the logic $L\mathrm{g}\lambda$ to prove properties of terms of the (simply typed) guarded λ-calculus, $\mathrm{g}\lambda$. This allowed proofs about coinductive types, but not in the integrated fashion supported by dependent type theories. Moreover it relied on types being "total", a property that in a dependently typed setting would entail a strong elimination rule for $\triangleright$, which would lead to inconsistency.

Sized types [45] have been combined with copatterns [2] as an alternative type-based approach for modular programming with coinductive types. This work is more mature than ours with respect to implementation and the demonstration of syntactic properties such as normalisation, and so further development of GDTT is essential to enable proper comparison. One advantage of GDTT is that the later modality is useful for examples beyond coinduction, and beyond the utility of sized types, such as the guarded recursive domain equations used to model program logics [82].

3.8 Conclusion and Future Work

We have described the dependent type theory GDTT. The examples we have detailed show that GDTT provides a setting for programming and proving with guarded recursive and coinductive types.

In future work we plan to investigate an intensional version of the type theory and construct a prototype implementation to allow us to experiment with larger examples. Preliminary work has suggested that the path type of cubical type theory [29] interacts better with the new constructs of GDTT than the ordinary Martin-Löf identity type.

Finally, we are investigating whether the generalisation of applicative functors [64] to apply over *dependent* function spaces, via delayed substitutions, might also apply to examples quite unconnected to the later modality.

Acknowledgements. This research was supported in part by the ModuRes Sapere Aude Advanced Grant and DFF-Research Project 1 Grant no. 4002-00442, both from The Danish Council for Independent Research for the Natural Sciences (FNU). Aleš Bizjak was supported in part by a Microsoft Research PhD grant.

3.A Overview of the appendix

Sec. 3.B contains type and term equalities of Fig. 3.3 in full detail. Sec. 3.C starting on page 85 contains detailed explanations of examples from Sec. 3.3 explaining how the rules of GDTT are used. Sec. 3.D starting on page 93 contains detailed explanations of examples with coinductive types. Sec. 3.E starting on page 95 contains a detailed derivation of the type isomorphism $\forall\kappa.A + B \cong \forall\kappa.A + \forall\kappa.B$ used in Sec. 3.4.

3.B Typing rules

Definitional type equalities:

$$\frac{\Gamma, \Gamma' \vdash_\Delta A \text{ type} \qquad \vdash_\Delta \xi[x \leftarrow t] : \Gamma \xrightarrow{\kappa} \Gamma', x : B}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow t].A \equiv \overset{\kappa}{\triangleright}\xi.A} \textsc{TyEq-}\triangleright\textsc{-Weak}$$

$$\frac{\begin{array}{c}\Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash_\Delta A \text{ type} \\ \vdash_\Delta \xi\,[x \leftarrow t, y \leftarrow u]\,\xi' : \Gamma \xrightarrow{\kappa} \Gamma', x : B, y : C, \Gamma'' \qquad x \text{ not free in } C\end{array}}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A \equiv \overset{\kappa}{\triangleright}\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A} \textsc{TyEq-}\triangleright\textsc{-Exch}$$

$$\frac{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t].A \text{ type}}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t].A \equiv \overset{\kappa}{\triangleright}\xi.A[t/x]} \textsc{TyEq-Force}$$

$$\frac{\Delta' \subseteq \Delta \qquad \vdash_{\Delta'} \kappa \qquad \Gamma, \Gamma' \vdash_\Delta A : \mathcal{U}_{\Delta'} \qquad \vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma'}{\Gamma \vdash_\Delta \mathsf{El}(\widehat{\triangleright}\kappa\,(\mathsf{next}^\kappa\,\xi.A)) \equiv \overset{\kappa}{\triangleright}\xi.\,\mathsf{El}(t)} \textsc{TyEq-El-}\triangleright$$

$$\frac{\vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma' \qquad \Gamma, \Gamma' \vdash_\Delta t : A \qquad \Gamma, \Gamma' \vdash_\Delta s : A}{\Gamma \vdash_\Delta \mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}(\mathsf{next}^\kappa\,\xi.t, \mathsf{next}^\kappa\,\xi.s) \equiv \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A\,(t,s)} \textsc{TyEq-}\triangleright$$

Definitional term equalities:

$$\frac{\Gamma, \Gamma' \vdash_\Delta u : A \qquad \vdash_\Delta \xi\,[x \leftarrow t] : \Gamma \xrightarrow{\kappa} \Gamma', x : B}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow t].u \equiv \mathsf{next}^\kappa\,\xi.u : \overset{\kappa}{\triangleright}\xi.A} \textsc{TmEq-Next-Weak}$$

$$\frac{\Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright}\xi.A}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow t].x \equiv t : \overset{\kappa}{\triangleright}\xi.A} \textsc{TmEq-Next-Var}$$

$$\frac{\begin{array}{c}\Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash_\Delta t : A \\ \vdash_\Delta \xi\,[x \leftarrow t, y \leftarrow u]\,\xi' : \Gamma \xrightarrow{\kappa} \Gamma', x : B, y : C, \Gamma'' \qquad x \text{ not free in } C\end{array}}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}^\kappa\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v : \overset{\kappa}{\triangleright}\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A} \textsc{TmEq-Next-Exch}$$

$$\frac{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t].u : \overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t].A}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t].u \equiv \mathsf{next}^\kappa\,\xi.u[t/x] : \overset{\kappa}{\triangleright}\xi.A[t/x]} \textsc{TmEq-Force}$$

$$\frac{\Gamma \vdash_\Delta \mathsf{fix}^\kappa\,x.t : A}{\Gamma \vdash_\Delta \mathsf{fix}^\kappa\,x.t \equiv t[\mathsf{next}^\kappa\,(\mathsf{fix}^\kappa\,x.t)/x] : A} \textsc{TmEq-Fix}$$

3.C Examples

In this section we provide detailed explanations of typing derivations of examples described in Sec. 3.3.

3.C.1 $\mathsf{zipWith}^\kappa$ preserves commutativity

The first proof is the simplest. We will define the standard $\mathsf{zipWith}^\kappa$ (zipWith) function on streams and show that if a binary function f is commutative, then so is $\mathsf{zipWith}^\kappa f$.

The $\mathsf{zipWith}^\kappa : (A \to B \to C) \to \mathsf{Str}^\kappa_A \to \mathsf{Str}^\kappa_B \to \mathsf{Str}^\kappa_C$ is defined by guarded recursion as

$$\begin{aligned}\mathsf{zipWith}^\kappa\, f \triangleq \mathsf{fix}^\kappa\, \varphi.\lambda(xs, ys : \mathsf{Str}^\kappa_A).\\ \mathsf{cons}^\kappa\,(f\,(\mathsf{hd}^\kappa\, xs)\,(\mathsf{hd}^\kappa\, ys))\,(\varphi \circledast \mathsf{tl}^\kappa\, xs \circledast \mathsf{tl}^\kappa\, ys)\end{aligned}$$

Note that none of the new generalised $\triangleright$ rules of GDTT are needed to type this function; this is a function on simple types.

Where we need dependent types is, of course, to state and prove properties. To prove our example, that commutativity of f implies commutativity of $\mathsf{zipWith}^\kappa f$, means we must show that the type

$$\begin{aligned}&\Pi(f : A \to A \to B).\,(\Pi(x, y : A).\mathsf{Id}_B\,(f\,x\,y, f\,y\,x)) \to\\ &\Pi\big(xs, ys : \mathsf{Str}^\kappa_A\big).\mathsf{Id}_{\mathsf{Str}^\kappa_B}\,(\mathsf{zipWith}^\kappa\, f\, xs\, ys, \mathsf{zipWith}^\kappa\, f\, ys\, xs).\end{aligned}$$

is inhabited. We will explain how to construct such a term, and why it is typeable in GDTT. Although this construction might appear complicated at first, the actual proof term that we construct will be as simple as possible.

Let $f : A \to A \to B$ be a function and say we have a term

$$c : \Pi(x, y : A).\mathsf{Id}_B\,(f\,x\,y, f\,y\,x)$$

witnessing commutativity of f. We now wish to construct a term of type

$$\Pi\big(xs, ys : \mathsf{Str}^\kappa_A\big).\mathsf{Id}_{\mathsf{Str}^\kappa_C}\,(\mathsf{zipWith}^\kappa\, f\, xs\, ys, \mathsf{zipWith}^\kappa\, f\, ys\, xs)$$

We do this by guarded recursion. To this end we assume

$$\varphi : {}^\kappa\!\triangleright\Big(\Pi\big(xs, ys : \mathsf{Str}^\kappa_A\big).\mathsf{Id}_{\mathsf{Str}^\kappa_B}\,(\mathsf{zipWith}^\kappa\, f\, xs\, ys, \mathsf{zipWith}^\kappa\, f\, ys\, xs)\Big)$$

and take $xs, ys : \mathsf{Str}^\kappa_A$. Using c (the proof that f is commutative) we first have $c\,(\mathsf{hd}^\kappa\, xs)\,(\mathsf{hd}^\kappa\, ys)$ of type

$$\mathsf{Id}_B\,(f\,(\mathsf{hd}^\kappa\, xs)\,(\mathsf{hd}^\kappa\, ys), f\,(\mathsf{hd}^\kappa\, ys)\,(\mathsf{hd}^\kappa\, xs))$$

and because we have by definition of $\mathsf{zipWith}^\kappa$

$$\mathsf{hd}^\kappa(\mathsf{zipWith}^\kappa f\, xs\, ys) \equiv f\,(\mathsf{hd}^\kappa xs)\,(\mathsf{hd}^\kappa ys)$$
$$\mathsf{hd}^\kappa(\mathsf{zipWith}^\kappa f\, ys\, xs) \equiv f\,(\mathsf{hd}^\kappa ys)\,(\mathsf{hd}^\kappa xs)$$

we see that $c\,(\mathsf{hd}^\kappa xs)\,(\mathsf{hd}^\kappa ys)$ has type

$$\mathsf{Id}_B(\mathsf{hd}^\kappa(\mathsf{zipWith}^\kappa f\, xs\, ys), \mathsf{hd}^\kappa(\mathsf{zipWith}^\kappa f\, ys\, xs)).$$

To show that the tails are equal we use the induction hypothesis φ. The terms $\mathsf{tl}^\kappa xs$ and $\mathsf{tl}^\kappa ys$ are of type ${}^\kappa\!\triangleright \mathsf{Str}^\kappa_A$, so we first have $\varphi \circledast \mathsf{tl}^\kappa xs$ of type

$${}^\kappa\!\triangleright[xs \leftarrow \mathsf{tl}^\kappa xs].\left(\Pi\left(ys : \mathsf{Str}^\kappa_A\right).\mathsf{Id}_{\mathsf{Str}^\kappa_C}\left(\begin{array}{l}\mathsf{zipWith}^\kappa f\, xs\, ys,\\ \mathsf{zipWith}^\kappa f\, ys\, xs\end{array}\right)\right)$$

Note the appearance of the generalised $\triangleright$, carrying a delayed substitution. Because the variable xs does not appear in ${}^\kappa\!\triangleright \mathsf{Str}^\kappa_A$ we may apply the weakening rule TmEq-Next-Weak to derive

$$\mathsf{tl}^\kappa ys : {}^\kappa\!\triangleright[xs \leftarrow \mathsf{tl}^\kappa xs].\,\mathsf{Str}^\kappa_A$$

Hence we may use the derived applicative rule to have $\varphi \circledast \mathsf{tl}^\kappa xs \circledast \mathsf{tl}^\kappa ys$ of type

$${}^\kappa\!\triangleright\left[\begin{array}{l} xs \leftarrow \mathsf{tl}^\kappa xs\\ ys \leftarrow \mathsf{tl}^\kappa ys\end{array}\right].\mathsf{Id}_{\mathsf{Str}^\kappa_C}(\mathsf{zipWith}^\kappa f\, xs\, ys, \mathsf{zipWith}^\kappa f\, ys\, xs)$$

and which is definitionally equal to the type

$$\mathsf{Id}_{{}^\kappa\!\triangleright \mathsf{Str}^\kappa_C}\left(\begin{array}{l}\mathsf{next}^\kappa\left[\begin{array}{l} xs \leftarrow \mathsf{tl}^\kappa xs\\ ys \leftarrow \mathsf{tl}^\kappa ys\end{array}\right].\mathsf{zipWith}^\kappa f\, xs\, ys,\\ \mathsf{next}^\kappa\left[\begin{array}{l} xs \leftarrow \mathsf{tl}^\kappa xs\\ ys \leftarrow \mathsf{tl}^\kappa ys\end{array}\right].\mathsf{zipWith}^\kappa f\, ys\, xs\end{array}\right).$$

We also compute

$$\begin{aligned}\mathsf{tl}^\kappa(\mathsf{zipWith}^\kappa f\, xs\, ys) &\equiv \mathsf{next}^\kappa(\mathsf{zipWith}^\kappa f) \circledast \mathsf{tl}^\kappa xs \circledast \mathsf{tl}^\kappa ys\\ &\equiv \mathsf{next}^\kappa\left[\begin{array}{l} xs \leftarrow \mathsf{tl}^\kappa xs\\ ys \leftarrow \mathsf{tl}^\kappa ys\end{array}\right].(\mathsf{zipWith}^\kappa f\, xs\, ys)\end{aligned}$$

and

$$\mathsf{tl}^\kappa(\mathsf{zipWith}^\kappa f\, ys\, xs) \equiv \mathsf{next}^\kappa\left[\begin{array}{l} ys \leftarrow \mathsf{tl}^\kappa ys\\ zs \leftarrow \mathsf{tl}^\kappa xs\end{array}\right].(\mathsf{zipWith}^\kappa f\, ys\, xs).$$

Using the exchange rule TmEq-Next-Exch we have the equality

$$\mathsf{next}^\kappa\left[\begin{array}{l} ys \leftarrow \mathsf{tl}^\kappa ys\\ xs \leftarrow \mathsf{tl}^\kappa xs\end{array}\right].(\mathsf{zipWith}^\kappa f\, xs\, ys) \equiv \mathsf{next}^\kappa\left[\begin{array}{l} xs \leftarrow \mathsf{tl}^\kappa xs\\ ys \leftarrow \mathsf{tl}^\kappa ys\end{array}\right].(\mathsf{zipWith}^\kappa f\, xs\, ys).$$

Putting it all together we have shown that the term $\varphi \circledast \mathsf{tl}^\kappa\, xs \circledast \mathsf{tl}^\kappa\, ys$ has type

$$\mathsf{Id}_{\overset{\kappa}{\triangleright}\mathsf{Str}_B^\kappa}(\mathsf{tl}^\kappa(\mathsf{zipWith}^\kappa\, f\, xs\, ys), \mathsf{tl}^\kappa(\mathsf{zipWith}^\kappa\, f\, ys\, xs))$$

which means that the term

$$\mathsf{fix}^\kappa\, \varphi.\lambda\left(xs, ys : \mathsf{Str}_A^\kappa\right).\mathsf{p}\eta\ (c(\mathsf{hd}^\kappa\, xs)(\mathsf{hd}^\kappa\, ys))\ (\varphi \circledast \mathsf{tl}^\kappa\, xs \circledast \mathsf{tl}^\kappa\, ys)$$

has type $\Pi\left(xs, ys : \mathsf{Str}_A^\kappa\right).\mathsf{Id}_{\mathsf{Str}_B^\kappa}(\mathsf{zipWith}^\kappa\, f\, xs\, ys, \mathsf{zipWith}^\kappa\, f\, ys\, xs)$.

Notice that the resulting proof term could not be simpler than it is. In particular, we do not have to write delayed substitutions in terms, but only in the intermediate types.

3.C.2 An example with covectors

The next example is more sophisticated, as it will involve programming and proving with a data type that, unlike streams, is dependently typed. In particular, we will see that the generalised later, carrying a delayed substitution, is necessary to type even the most elementary programs.

Covectors are to colists (potentially infinite lists) as vectors are to lists. To define guarded covectors we first need guarded co-natural numbers. This is the type satisfying

$$\mathsf{CoIN}^\kappa \equiv \mathbf{1} + \overset{\kappa}{\triangleright}\mathsf{CoIN}^\kappa.$$

where binary sums are encoded in the type theory in a standard way. The definition in GDTT is $\mathsf{CoIN}^\kappa \triangleq \mathsf{El}\left(\mathsf{fix}^\kappa\, \varphi.(\widehat{\mathbf{1} + \triangleright}\kappa\varphi)\right)$.

Using CoIN^κ we define the type of covectors of type A, written CoVec_A^κ, as a CoIN^κ-indexed type satisfying

$$\mathsf{CoVec}_A^\kappa(\mathsf{inl}\,\langle\rangle) \equiv \mathbf{1}$$
$$\mathsf{CoVec}_A^\kappa(\mathsf{inr}(\mathsf{next}^\kappa\, m)) \equiv A \times \overset{\kappa}{\triangleright}(\mathsf{CoVec}_A^\kappa\, m)$$

In GDTT we first define $\widehat{\mathsf{CoVec}}_A^\kappa$

$$\begin{aligned}\widehat{\mathsf{CoVec}}_A^\kappa \triangleq \mathsf{fix}^\kappa\, \varphi.\lambda(n : \mathsf{CoIN}^\kappa).\, &\mathsf{case}\, n\, \mathsf{of}\\ &\mathsf{inl}\, u \Rightarrow \widehat{\mathbf{1}}\\ &\mathsf{inr}\, m \Rightarrow A\,\widehat{\times\triangleright}\,\kappa(\varphi \circledast m).\end{aligned}$$

and then $\mathsf{CoVec}_A^\kappa\, n \triangleq \mathsf{El}(\widehat{\mathsf{CoVec}}_A^\kappa\, n)$. In the examples we will not distinguish between CoVec_A^κ and $\widehat{\mathsf{CoVec}}_A^\kappa$. In the above φ has type $\overset{\kappa}{\triangleright}(\mathsf{CoIN}^\kappa \to \mathcal{U}_{\Delta,\kappa})$ and inside the branches, u has type $\mathbf{1}$ and m has type $\overset{\kappa}{\triangleright}\mathsf{CoIN}^\kappa$, which is evident from the definition of CoIN^κ. As an example of covectors, we define ones of type

$\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\, \mathsf{CoVec}^\kappa_{\mathbb{N}}\, n$ which produces a covector of any length consisting only of ones:

$$\begin{aligned}\mathsf{ones} \triangleq \mathsf{fix}^\kappa\, \varphi.\lambda(n : \mathsf{Co}\mathbb{N}^\kappa).\, \mathsf{case}\, n\, \mathsf{of}\\ \mathsf{inl}\, u \Rightarrow \mathsf{inl}\, \langle\rangle\\ \mathsf{inr}\, m \Rightarrow \langle 1, \varphi \circledast m\rangle.\end{aligned}$$

When checking the type of this program, we need the generalised later. The type of the recursive call is $\overset{\kappa}{\triangleright}(\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\, \mathsf{CoVec}^\kappa_{\mathbb{N}}\, n)$, the type of m is $\overset{\kappa}{\triangleright}\mathsf{Co}\mathbb{N}^\kappa$, and therefore the type of the subterm $\varphi \circledast m$ must be

$$\overset{\kappa}{\triangleright}[n \leftarrow m].\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\, \mathsf{CoVec}^\kappa_{\mathbb{N}}\, n.x$$

We now aim to define the function map on covectors and show that it preserves composition. Given two types A and B the map function has type

$$\mathsf{map} : (A \to B) \to \Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\, \mathsf{CoVec}^\kappa_A\, n \to \mathsf{CoVec}^\kappa_B\, n.$$

and is defined by guarded recursion as

$$\begin{aligned}\mathsf{map}\, f \triangleq\ & \mathsf{fix}^\kappa\, \varphi.\lambda(n : \mathsf{Co}\mathbb{N}^\kappa).\\ & \mathsf{case}\, n\, \mathsf{of}\\ & \mathsf{inl}\, u \Rightarrow \lambda(x : 1).x\\ & \mathsf{inr}\, m \Rightarrow \begin{array}{l}\lambda\Big(p : A \times \overset{\kappa}{\triangleright}[n \leftarrow m].(\mathsf{CoVec}^\kappa_A\, n)\Big).\\ \quad \langle f\, (\pi_1 p), \varphi \circledast m \circledast (\pi_2 p)\rangle\end{array}\end{aligned}$$

Let us see why the definition has the correct type. First, the types of subterms are

$$\begin{aligned}\varphi &: \overset{\kappa}{\triangleright}(\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\, \mathsf{CoVec}^\kappa_A\, n \to \mathsf{CoVec}^\kappa_B\, n)\\ u &: \mathbf{1}\\ m &: \overset{\kappa}{\triangleright}\mathsf{Co}\mathbb{N}^\kappa\end{aligned}$$

Let $C = \mathsf{CoVec}^\kappa_A\, n \to \mathsf{CoVec}^\kappa_B\, n$, and write $C(t)$ for $C[t/n]$. By the definition of CoVec^κ_A and CoVec^κ_B we have $C(\mathsf{inl}\, u) \equiv \mathbf{1} \to \mathbf{1}$, and so $\lambda(x : \mathbf{1}).x$ has type $C(\mathsf{inl}\, u)$.

By the definition of CoVec^κ_A we have

$$\begin{aligned}\mathsf{CoVec}^\kappa_A(\mathsf{inr}\, m) &\equiv A \times \mathsf{El}\Big(\widehat{\triangleright}\kappa(\mathsf{next}^\kappa(\mathsf{CoVec}^\kappa_A) \circledast m)\Big)\\ &\equiv A \times \overset{\kappa}{\triangleright}[n \leftarrow m].\Big(\mathsf{CoVec}^\kappa_A\, n\Big)\end{aligned}$$

and analogously for $\mathsf{CoVec}^\kappa_B(\mathsf{inr}\, m)$. Hence the type $C(\mathsf{inr}\, m)$ is convertible to

$$\Big(A \times \overset{\kappa}{\triangleright}[n \leftarrow m].\Big(\mathsf{CoVec}^\kappa_A\, n\Big)\Big) \to \Big(B \times \overset{\kappa}{\triangleright}[n \leftarrow m].\Big(\mathsf{CoVec}^\kappa_B\, n\Big)\Big).$$

Further, using the derived applicative rule we have

$$\varphi \circledast m : \overset{\kappa}{\triangleright}[n \leftarrow m].C(n)$$

and because $\pi_2 p$ in the second branch has type

$$\overset{\kappa}{\triangleright}[n \leftarrow m].(\mathsf{CoVec}^{\kappa}_{A}\, n)$$

we may use the (simple) applicative rule again to get

$$\varphi \circledast m \circledast (\pi_2 p) : \overset{\kappa}{\triangleright}[n \leftarrow m].(\mathsf{CoVec}^{\kappa}_{B}\, n)$$

which allows us to type

$$\lambda\left(p : A \times \overset{\kappa}{\triangleright}[n \leftarrow m].(\mathsf{CoVec}^{\kappa}_{A}\, n)\right).\langle f\,(\pi_1 p), \varphi \circledast m \circledast \pi_2(p)\rangle$$

with type $C(\mathsf{inr}\, m)$. Notice that we have made essential use of the more general applicative rule to apply $\varphi \circledast m$ to $\pi_2 p$. Using the strong (dependent) elimination rule for binary sums we can type the whole case construct with type $C(n)$, which is what we need to give map the desired type.

Now we will show that map so defined satisfies a basic property, namely that it preserves composition in the sense that the type (in the context where we have types A, B and C)

$$\begin{aligned}\Pi(f : A \to B)(g : B \to C)(n : \mathsf{CoIN}^{\kappa})(xs : \mathsf{CoVec}^{\kappa}_{A}\, n).\\ \mathsf{Id}_{\mathsf{CoVec}^{\kappa}_{C}\, n}(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \mathsf{map}(g \circ f)\, n\, xs)\end{aligned} \tag{3.8}$$

is inhabited. The proof is, of course, by Löb induction.

First we record some definitional equalities which follow directly by unfolding the definitions

$$\begin{aligned}\mathsf{map}\, f\,(\mathsf{inl}\, u)\, x &\equiv x\\ \mathsf{map}\, f\,(\mathsf{inr}\, m)\, xs &\equiv \langle f\,(\pi_1 xs), \mathsf{next}^{\kappa}(\mathsf{map}\, f) \circledast m \circledast \pi_2(xs)\rangle\\ &\equiv \langle f(\pi_1 xs), \mathsf{next}^{\kappa}\begin{bmatrix} n \leftarrow m \\ ys \leftarrow \pi_2 xs \end{bmatrix}.(\mathsf{map}\, f\, n\, ys)\rangle\end{aligned}$$

and so iterating these two equalities we get

$$\begin{aligned}\mathsf{map}\, g\,(\mathsf{inl}\, u)\,(\mathsf{map}\, f\,(\mathsf{inl}\, u)\, x) &\equiv x\\ \mathsf{map}\, g\,(\mathsf{inr}\, m)\,(\mathsf{map}\, f\,(\mathsf{inr}\, m)\, xs) &\equiv \langle g(f\,(\pi_1 xs)), s\rangle\end{aligned}$$

where s is the term

$$\mathsf{next}^{\kappa}\begin{bmatrix} n \leftarrow m \\ zs \leftarrow \mathsf{next}^{\kappa}\begin{bmatrix} n \leftarrow m \\ ys \leftarrow \pi_2 xs \end{bmatrix}.(\mathsf{map}\, f\, n\, ys) \end{bmatrix}.(\mathsf{map}\, g\, n\, zs)$$

which is convertible, by the rule TmEq-Force, to the term

$$\mathsf{next}^\kappa\left[\begin{array}{l} n \leftarrow m \\ ys \leftarrow \pi_2 xs \end{array}\right].(\mathsf{map}\, g\, n\, (\mathsf{map}\, f\, n\, ys)).$$

Similarly we have

$$\mathsf{map}(g \circ f)\,(\mathsf{inl}\, u)\, x \equiv x$$

and $\mathsf{map}(g \circ f)\,(\mathsf{inr}\, m)\, xs$ convertible to

$$\left\langle g(f(\pi_1 xs)), \mathsf{next}^\kappa\left[\begin{array}{l} n \leftarrow m \\ ys \leftarrow \pi_2 xs \end{array}\right].(\mathsf{map}(g \circ f)\, n\, ys)\right\rangle.$$

Now let us get back to proving property (3.8). Take $f : A \to B$, $g : B \to C$ and assume

$$\varphi : {}^\kappa\!\triangleright \Pi(n : \mathsf{CoIN}^\kappa)(xs : \mathsf{CoVec}^\kappa_A\, n).\mathsf{Id}_{\mathsf{CoVec}^\kappa_C\, n}\,(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \mathsf{map}(g \circ f)\, n\, xs)$$

We take $n : \mathsf{CoIN}^\kappa$ and write

$$P(n) = \Pi(xs : \mathsf{CoVec}^\kappa_A\, n).\mathsf{Id}_{\mathsf{CoVec}^\kappa_C\, n}\,(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \mathsf{map}(g \circ f)\, n\, xs).$$

Then similarly as in the definition of map and the definitional equalities for map above we compute

$$P(\mathsf{inl}\, u) \equiv \Pi(xs : 1).\mathsf{Id}_1\,(xs, xs)$$

and so we have $\lambda(xs : 1).\mathsf{r}_1\, xs$ of type $P(\mathsf{inl}\, u)$.

The other branch (when $n = \mathsf{inr}\, m$) is of course a bit more complicated. As before we have

$$\mathsf{CoVec}^\kappa_A(\mathsf{inr}\, m) \equiv A \times {}^\kappa\!\triangleright [n \leftarrow m].\, \mathsf{CoVec}^\kappa_A\, n \tag{3.9}$$

So take xs of type $\mathsf{CoVec}^\kappa_A(\mathsf{inr}\, m)$. We need to construct a term of type

$$\mathsf{Id}_{\mathsf{CoVec}^\kappa_C\, n}\,(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \mathsf{map}(g \circ f)\, n\, xs).$$

First we have $\mathsf{r}_C\, g(f(\pi_1 xs))$ of type $\mathsf{Id}_C\,(g(f(\pi_1 xs)), g(f(\pi_1 xs)))$. Then because m is of type ${}^\kappa\!\triangleright \mathsf{CoIN}^\kappa$ we can use the induction hypothesis φ to get $\varphi \circledast m$ of type

$${}^\kappa\!\triangleright [n \leftarrow m].\Pi(xs : \mathsf{CoVec}^\kappa_A\, n).\mathsf{Id}_{\mathsf{CoVec}^\kappa_C\, n}\,(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \mathsf{map}(g \circ f)\, n\, xs).$$

Using (3.9) we have $\pi_2 xs$ of type ${}^\kappa\!\triangleright [n \leftarrow m].\, \mathsf{CoVec}^\kappa_A\, n$ and so we can use the applicative rule again to give $\varphi \circledast m \circledast \pi_2 xs$ the type

$${}^\kappa\!\triangleright\left[\begin{array}{l} n \leftarrow m \\ xs \leftarrow \pi_2 xs \end{array}\right].\mathsf{Id}_{\mathsf{CoVec}^\kappa_C\, n}\left(\begin{array}{l} \mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \\ \mathsf{map}(g \circ f)\, n\, xs \end{array}\right)$$

which by the rule TyEq-▷ is the same as

$$\mathsf{Id}_D\left(\begin{array}{l}\mathsf{next}^\kappa\left[\begin{array}{l} n \leftarrow m \\ xs \leftarrow \pi_2 xs\end{array}\right].(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs)), \\ \mathsf{next}^\kappa\left[\begin{array}{l} n \leftarrow m \\ xs \leftarrow \pi_2 xs\end{array}\right].(\mathsf{map}(g \circ f)\, n\, xs)\end{array}\right)$$

where D is the type $\overset{\kappa}{\triangleright}[n \leftarrow m].\,\mathsf{CoVec}^\kappa_C\, n$. Thus we can give to the term

$$\lambda(xs : \mathsf{CoVec}^\kappa_A(\mathsf{inr}\, m)).\,\mathsf{p}\eta\,(\mathsf{r}_C\, g(f(\pi_1 xs)))\,(\varphi \circledast m \circledast \pi_2 xs)$$

the type $P(\mathsf{inr}\, m)$. Using the dependent elimination rule for binary sums we get the final proof of property (3.8) as the term

$$\begin{array}{l}\lambda(f : A \to B)(g : B \to C).\,\mathsf{fix}^\kappa\, \varphi.\lambda(n : \mathsf{CoIN}^\kappa). \\ \mathsf{case}\, n\, \mathsf{of} \\ \mathsf{inl}\, u \Rightarrow \lambda(xs : 1).\mathsf{r}_1\, xs \\ \mathsf{inr}\, m \Rightarrow \lambda(xs : \mathsf{CoVec}^\kappa_A(\mathsf{inr}\, m)).\,\mathsf{p}\eta\,(\mathsf{r}_C\, g(f(\pi_1 xs)))\,(\varphi \circledast m \circledast \pi_2 xs)\end{array}$$

which is as simple as could be expected.

3.C.3 Lifting predicates to streams

Let $P : A \to \mathcal{U}_\Delta$ be a predicate on type A and κ a clock variable not in Δ. We can define a lifting of this predicate to a predicate P^κ on streams of elements of type A. The idea is that $P^\kappa xs$ will hold precisely when P holds for all elements of the stream. However we do not have access to all the element of the stream at the same time. As such we will have $P^\kappa xs$ if P holds for the first element of the stream xs now, and P holds for the second element of the stream xs one time step later, and so on. The precise definition uses guarded recursion:

$$\begin{array}{l}P^\kappa : \mathsf{Str}^\kappa_A \to \mathcal{U}_{\Delta,\kappa} \\ P^\kappa \triangleq \mathsf{fix}^\kappa\, \varphi.\lambda\left(xs : \mathsf{Str}^\kappa_A\right).P\,(\mathsf{hd}^\kappa\, xs)\,\widehat{\times}\widehat{\triangleright}\kappa\,(\varphi \circledast \mathsf{tl}^\kappa\, xs).\end{array}$$

In the above term the subterm φ has type $\overset{\kappa}{\triangleright}\left(\mathsf{Str}^\kappa_A \to \mathcal{U}_{\Delta,\kappa}\right)$ and so because $\mathsf{tl}^\kappa\, xs$ has type $\overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A$ we may form $\varphi \circledast \mathsf{tl}^\kappa\, xs$ of type $\overset{\kappa}{\triangleright}\mathcal{U}_{\Delta,\kappa}$ and so finally $\widehat{\triangleright}\kappa(\varphi \circledast \mathsf{tl}^\kappa\, xs)$ has type $\mathcal{U}_{\Delta,\kappa}$ as needed.

To see that this makes sense, we have for a stream $xs : \mathsf{Str}^\kappa_A$

$$\mathsf{El}\,(P^\kappa\, xs) \equiv \mathsf{El}\,(P\,(\mathsf{hd}^\kappa\, xs)) \times \mathsf{El}\,(\widehat{\triangleright}\kappa\,(\mathsf{next}^\kappa\, P^\kappa \circledast \mathsf{tl}^\kappa\, xs)).$$

Using delayed substitution rules we have

$$\mathsf{next}^\kappa\, P^\kappa \circledast \mathsf{tl}^\kappa\, xs \equiv \mathsf{next}^\kappa\,[xs \leftarrow \mathsf{tl}^\kappa\, xs].\,(P^\kappa\, xs)$$

which gives rise to the type equality

$$\mathsf{El}(\widehat{\triangleright}\kappa\,\mathsf{next}^\kappa\,P^\kappa\,\circledast\,\mathsf{tl}^\kappa\,xs) \equiv \mathsf{El}\,(\widehat{\triangleright}\kappa\,\mathsf{next}^\kappa\,[xs \leftarrow \mathsf{tl}^\kappa\,xs].\,(P^\kappa\,xs)).$$

Finally, the type equality rule TyEq-El-$\triangleright$ gives us

$$\mathsf{El}\,(\widehat{\triangleright}\kappa\,\mathsf{next}^\kappa\,[xs \leftarrow \mathsf{tl}^\kappa\,xs].\,(P^\kappa\,xs)) \equiv {}^{\kappa}\!\triangleright[xs \leftarrow \mathsf{tl}^\kappa\,xs].\,\mathsf{El}(P^\kappa\,xs).$$

All of these together then give us the type equality

$$\mathsf{El}\,(P^\kappa\,xs) \equiv \mathsf{El}(P\,(\mathsf{hd}^\kappa\,xs)) \times {}^{\kappa}\!\triangleright[xs \leftarrow \mathsf{tl}^\kappa\,xs].\,\mathsf{El}(P^\kappa\,xs).$$

And so if $xs = \mathsf{cons}^\kappa\,x\,(\mathsf{next}^\kappa\,ys)$ we can further simplify, using rule TyEq-Force, to get

$${}^{\kappa}\!\triangleright[xs \leftarrow \mathsf{next}^\kappa\,ys].\,\mathsf{El}(P^\kappa\,xs) \equiv {}^{\kappa}\!\triangleright(\mathsf{El}(P^\kappa\,xs)[ys/xs]) \equiv {}^{\kappa}\!\triangleright\mathsf{El}(P^\kappa\,ys)$$

which then gives $\mathsf{El}(P^\kappa xs) \equiv \mathsf{El}\,(P\,x) \times {}^{\kappa}\!\triangleright\mathsf{El}\,(P^\kappa\,ys)$ which is in accordance with the motivation given above.

Because P^κ is defined by guarded recursion, we prove its properties by Löb induction. In particular, we may prove that if P holds on A then P^κ holds on Str^κ_A, i.e., that the type

$$(\Pi(x : A).\,\mathsf{El}\,(P\,x)) \to \left(\Pi\left(xs : \mathsf{Str}^\kappa_A\right).\,\mathsf{El}\,(P^\kappa\,xs)\right)$$

is inhabited (in a context where we have a type A and a predicate P). Take $p : \Pi(x : A).\,\mathsf{El}\,(P\,x)$, and since we are proving by Löb induction we assume the induction hypothesis later

$$\varphi : {}^{\kappa}\!\triangleright\left(\Pi\left(xs : \mathsf{Str}^\kappa_A\right).\,\mathsf{El}\,(P^\kappa\,xs)\right).$$

Let $xs : \mathsf{Str}^\kappa_A$ be a stream. By definition of P^κ we have the type equality

$$\mathsf{El}(P^\kappa xs) \equiv \mathsf{El}\,(P\,\mathsf{hd}^\kappa\,xs) \times {}^{\kappa}\!\triangleright[xs \leftarrow \mathsf{tl}^\kappa\,xs].\,\mathsf{El}\,(P^\kappa\,xs)$$

Applying p to $\mathsf{hd}^\kappa\,xs$ gives us the first component

$$p(\mathsf{hd}^\kappa\,xs) : \mathsf{El}\,(P\,(\mathsf{hd}^\kappa\,xs))$$

and applying the induction hypothesis φ we have

$$\varphi\,\circledast\,\mathsf{tl}^\kappa\,xs : {}^{\kappa}\!\triangleright[xs \leftarrow \mathsf{tl}^\kappa\,xs].\,\mathsf{El}(P^\kappa\,xs)$$

Thus combining this with the previous term we have the proof of the lifting property as the term

$$\begin{aligned}&\lambda\,(p : \Pi(x : A).\,\mathsf{El}\,(P\,x)).\\ &\quad \mathsf{fix}^\kappa\,\varphi.\lambda\left(xs : \mathsf{Str}^\kappa_A\right)\langle p\,(\mathsf{hd}^\kappa\,xs), \varphi\,\circledast\,\mathsf{tl}^\kappa\,xs\rangle.\end{aligned}$$

3.D Example programs with coinductive types

Let A be some small type in clock context Δ and κ, a fresh clock variable. Let $\mathsf{Str}_A = \forall\kappa.\,\mathsf{Str}_A^\kappa$. We can define head, tail and cons functions

$$\begin{array}{ll} \mathsf{hd} : \mathsf{Str}_A \to A & \mathsf{tl} : \mathsf{Str}_A \to \mathsf{Str}_A \\ \mathsf{hd} \triangleq \lambda xs.\,\mathsf{hd}^{\kappa_0}\,(xs[\kappa_0]) & \mathsf{tl} \triangleq \lambda xs.\,\mathsf{prev}\,\kappa.\,\mathsf{tl}^\kappa\,(xs[\kappa]) \end{array}$$

$$\begin{array}{l} \mathsf{cons} : A \to \mathsf{Str}_A \to \mathsf{Str}_A \\ \mathsf{cons} \triangleq \lambda x.\lambda xs.\Lambda\kappa.\,\mathsf{cons}^\kappa\,x\,(\mathsf{next}^\kappa\,(xs[\kappa])). \end{array}$$

With these we can define the *acausal* 'every other' function eo^κ that removes every second element of the input stream. This is acausal because the second element of the output stream is the third element of the input. Therefore to type the function we need to have the input stream always available, necessitating the use clock quantification. The function eo^κ is

$$\begin{array}{l} \mathsf{eo}^\kappa : \mathsf{Str}_A \to \mathsf{Str}_A^\kappa \\ \mathsf{eo}^\kappa \triangleq \mathsf{fix}^\kappa\,\varphi.\lambda\,(xs : \mathsf{Str}_A). \\ \qquad\qquad \mathsf{cons}^\kappa(\mathsf{hd}\,xs)\,(\varphi\,\textcircled{\kappa}\,\mathsf{next}^\kappa\,((\mathsf{tl}(\mathsf{tl}\,xs)))). \end{array}$$

i.e., we return the head immediately and then recursively call the function on the stream with the first two elements removed. Note that the result is a *guarded* stream, but we can easily strengthen it and define eo of type $\mathsf{Str}_A \to \mathsf{Str}_A$ as $\mathsf{eo} \triangleq \lambda xs.\Lambda\kappa.\,\mathsf{eo}^\kappa\,xs$.

A more interesting type is the type of covectors, which is a refinement of the guarded type of covectors defined in Sec. 3.3. First we define the type of co-natural numbers CoℕN as

$$\mathsf{CoℕN} = \forall\kappa.\,\mathsf{CoℕN}^\kappa.$$

It is easy to define $\overline{0}$ and $\overline{\mathsf{succ}}$ as

$$\begin{array}{ll} \overline{0} : \mathsf{CoℕN} & \overline{\mathsf{succ}} : \mathsf{CoℕN} \to \mathsf{CoℕN} \\ \overline{0} \triangleq \Lambda\kappa.\,\mathsf{inl}\,\langle\rangle & \overline{\mathsf{succ}} \triangleq \lambda n.\Lambda\kappa.\,\mathsf{inr}\,(\mathsf{next}^\kappa\,(n[\kappa]))^{\cdot} \end{array}$$

Next, we will use type isomorphisms to define a transport function $\mathsf{com}^{\mathsf{CoℕN}}$ of type $\mathsf{com}^{\mathsf{CoℕN}} : \mathsf{CoℕN} \to 1 + \mathsf{CoℕN}$ as

$$\begin{array}{l} \mathsf{com}^{\mathsf{CoℕN}} \triangleq \lambda n.\,\mathsf{case}\,\mathsf{com}^+\,n\,\mathsf{of} \\ \qquad\qquad \mathsf{inl}\,u \Rightarrow \mathsf{inl}\,u[\kappa_0] \\ \qquad\qquad \mathsf{inr}\,n \Rightarrow \mathsf{inr}\,\mathsf{prev}\,\kappa.n[\kappa] \end{array}$$

This function satisfies term equalities

$$\mathsf{com}^{\mathsf{CoIN}}\,\overline{0} \equiv \mathsf{inl}\,\langle\rangle \qquad\qquad \mathsf{com}^{\mathsf{CoIN}}(\overline{\mathsf{succ}}\,n) \equiv \mathsf{inr}\,n. \tag{3.10}$$

Using this we can define type of covectors CoVec_A as

$$\mathsf{CoVec}_A\ n \triangleq \forall\kappa.\,\mathsf{CoVec}_A^{\kappa}\ n$$

where $\mathsf{CoVec}_A^{\kappa} : \mathsf{CoIN} \to \mathcal{U}_{\Delta,\kappa}$ is the term

$$\begin{aligned}\mathsf{fix}^{\kappa}\,\varphi.\lambda\,(n : \mathsf{CoIN}).\,\mathsf{case}\,\mathsf{com}^{\mathsf{CoIN}}\,n\,\mathsf{of}\\ \mathsf{inl}\,\_ &\Rightarrow \widehat{1}\\ \mathsf{inr}\,n &\Rightarrow A\widehat{\times\triangleright}\kappa\,(\varphi\,\circledast\,(\mathsf{next}^{\kappa}\,n)).\end{aligned}$$

Notice the use of $\mathsf{com}^{\mathsf{CoIN}}$ to transport n of type CoIN to a term of type $1+\mathsf{CoIN}$ which we can case analyse. To see that this type satisfies the correct type equalities we need some auxiliary term equalities which follow from the way we have defined the terms.

Using term equalities (3.4) and (3.5) we can derive the (almost) expected type equalities

$$\begin{aligned}\mathsf{CoVec}_A\,\overline{0} &\equiv \forall\kappa.1\\ \mathsf{CoVec}_A\,(\overline{\mathsf{succ}}\,n) &\equiv \forall\kappa.\left(A \times^{\kappa}\triangleright\,(\mathsf{CoVec}^{\kappa}\,n)\right)\end{aligned} \tag{3.11}$$

and using the type isomorphisms we can extend these type equalities to type isomorphisms

$$\begin{aligned}\mathsf{CoVec}_A\,\overline{0} &\cong 1\\ \mathsf{CoVec}_A\,(\overline{\mathsf{succ}}\,n) &\cong A \times \mathsf{CoVec}_A\ n\end{aligned}$$

which are the expected type properties of the covector type.

A simple function we can define is the tail function

$$\begin{aligned}&\mathsf{tl} : \mathsf{CoVec}_A(\overline{\mathsf{succ}}\,n) \to \mathsf{CoVec}_A\\ &\mathsf{tl} \triangleq \lambda v.\,\mathsf{prev}\,\kappa.\pi_2\,(v[\kappa]).\end{aligned}$$

Note that we have used (3.11) to ensure that tl is type correct.

Next, we define the map function on covectors.

$$\begin{aligned}&\mathsf{map} : (A \to B) \to \Pi(n : \mathsf{CoIN}).\,\mathsf{CoVec}_A\,n \to \mathsf{CoVec}_B\,n\\ &\mathsf{map}\,f = \lambda n.\lambda xs.\Lambda\kappa.\,\mathsf{map}^{\kappa}\,f\,n\,(xs[\kappa])\end{aligned}$$

where map^{κ} is the function of type

$$\mathsf{map}^{\kappa} : (A \to B) \to \Pi(n : \mathsf{CoIN}).\,\mathsf{CoVec}_A^{\kappa}\,n \to \mathsf{CoVec}_B^{\kappa}\,n$$

defined as

$$\begin{aligned}\lambda f.\,\mathsf{fix}^{\kappa}\,\varphi.\lambda n.\,\mathsf{case}\,\mathsf{com}^{\mathsf{CoIN}}\,n\,\mathsf{of}\\ \mathsf{inl}\_ \Rightarrow \lambda v.v\\ \mathsf{inr}\,n \Rightarrow \lambda v.\langle f(\pi_1 v), \varphi \circledast (\mathsf{next}^{\kappa}\,n) \circledast \pi_2(v)\rangle.\end{aligned}$$

Let us see that this has the correct type. Let $D_A(x)$ (and analogously $D_B(x)$) be the type

$$\begin{aligned}D_A(x) \quad \triangleq \quad & \mathsf{case}\,x\,\mathsf{of}\\ & \mathsf{inl}\_ \Rightarrow \widehat{1}\\ & \mathsf{inr}\,n \Rightarrow A\widehat{\times\triangleright}\kappa\left(\left(\mathsf{next}^{\kappa}\,\mathsf{CoVec}_A^{\kappa}\right) \circledast (\mathsf{next}^{\kappa}\,n)\right).\end{aligned}$$

where x is of type $1 + \mathsf{CoIN}$. Using this abbreviation we can write the type of map^{κ} as

$$(A \to B) \to \Pi(n : \mathsf{CoIN}).D_A(\mathsf{com}^{\mathsf{CoIN}}\,n) \to D_B(\mathsf{com}^{\mathsf{CoIN}}\,n).$$

Using this it is straightforward to show, using the dependent elimination rule for sums, as we did in Sec. 3.3, that map^{κ} has the correct type. Indeed we have $D_A(\mathsf{inl}\,z) \equiv 1$ and $D_A(\mathsf{inr}\,n) \equiv A \times \overset{\kappa}{\triangleright}(\mathsf{CoVec}_A\,n)$.

3.E Type isomorphisms in detail

- If $\kappa \notin A$ then $\forall\kappa.A \cong A$. The terms are $\lambda x.x[\kappa_0]$ and $\lambda x.\Lambda\kappa.x$. The rule TmEq-∀-fresh is crucially needed to show that they constitute a type isomorphism.
- If $\kappa \notin A$ then $\forall\kappa.\Pi(x : A).B \cong \Pi(x : A).\forall\kappa.B$. The terms are

  $$\lambda z.\lambda x.\Lambda\kappa.z[\kappa]\,x$$

  of type $\forall\kappa.\Pi(x : A).B \to \Pi(x : A).\forall\kappa.B$ and

  $$\lambda z.\Lambda\kappa.\lambda x.(z\,x)[\kappa]$$

  of type $\Pi(x : A).\forall\kappa.B \to \forall\kappa.\Pi(x : A).B$.
- $\forall\kappa.\Sigma(x : A)\,B \cong \Sigma(y : \forall\kappa.A)(\forall\kappa.B[y[\kappa]/x])$. The terms are

  $$\lambda z.\langle\Lambda\kappa.\pi_1(z[\kappa]), \Lambda\kappa.\pi_2(z[\kappa])\rangle$$

  of type

  $$\forall\kappa.\Sigma(x : A)\,B \to \Sigma(y : \forall\kappa.A)(\forall\kappa.B[y[\kappa]/x])$$

  and

  $$\lambda z.\Lambda\kappa.\langle(\pi_1\,z)[\kappa], (\pi_2\,z)[\kappa]\rangle$$

  of the converse type.

- $\forall\kappa.A \cong \forall\kappa.\overset{\kappa}{\triangleright}A$. The terms are

$$\lambda z.\Lambda\kappa.\,\mathsf{next}^{\kappa}(z[\kappa])$$

of type $\forall\kappa.A \to \forall\kappa.\overset{\kappa}{\triangleright}A$ and

$$\lambda z.\,\mathsf{prev}\,\kappa.\,(z[\kappa])$$

of the converse type. The β and η rules for $\mathsf{prev}\,\kappa.$ ensure that this pair of functions constitutes an isomorphism.

Using these isomorphisms we can construct an additional type isomorphism witnessing that $\forall\kappa$ commutes with binary sums. Recall that we encode binary coproducts using Σ-types and universes in the standard way. Given two codes $\widehat{A}$ and $\widehat{B}$ in some universe $\mathcal{U}_\Delta$ we define

$$\begin{aligned}&\widehat{A}\widehat{+}\widehat{B} : \mathcal{U}_\Delta\\ &\widehat{A}\widehat{+}\widehat{B} \triangleq \Sigma\,(b : \mathbf{B})\,\mathsf{if}\,b\,\mathsf{then}\,\widehat{A}\,\mathsf{else}\,\widehat{B}\end{aligned}$$

and we write $A + B$ for $\mathsf{El}\left(\widehat{A}\widehat{+}\widehat{B}\right)$. Suppose that $\Delta' \subseteq \Delta$ and κ is a clock variable not in Δ. Suppose that $\Gamma \vdash_\Delta$ and that we have two codes $\widehat{A}, \widehat{B}$ satisfying

$$\Gamma \vdash_{\Delta,\kappa} \widehat{A} : \mathcal{U}_{\Delta',\kappa} \qquad\qquad \Gamma \vdash_{\Delta,\kappa} \widehat{B} : \mathcal{U}_{\Delta',\kappa}$$

We start with an auxiliary function $\mathsf{com}^{\mathsf{if}}$. Let b be some term of type $\mathbf{B}$. We then define

$$\begin{aligned}&\mathsf{com}^{\mathsf{if}}_b : \forall\kappa.\,\mathsf{El}\left(\mathsf{if}\,b\,\mathsf{then}\,\widehat{A}\,\mathsf{else}\,\widehat{B}\right)\\ &\qquad\qquad \to \mathsf{El}\left(\mathsf{if}\,b\,\mathsf{then}\,\widehat{\forall}\,\Lambda\kappa.\widehat{A}\,\mathsf{else}\,\widehat{\forall}\,\Lambda\kappa.\widehat{B}\right)\\ &\mathsf{com}^{\mathsf{if}}_b \triangleq \mathsf{if}\,b\,\mathsf{then}\,\lambda x.x\,\mathsf{else}\,\lambda x.x\end{aligned}$$

which is typeable due to the strong elimination rule for $\mathbf{B}$.

We now define the function com^{+}

$$\begin{aligned}&\mathsf{com}^{+} : \forall\kappa.(A + B) \to \forall\kappa.A + \forall\kappa.B\\ &\mathsf{com}^{+} \triangleq \lambda z.\left\langle \pi_1\,(z[\kappa_0]), \mathsf{com}^{\mathsf{if}}_{\pi_1\,(z[\kappa_0])}\,(\Lambda\kappa.\pi_2\,(z[\kappa]))\right\rangle.\end{aligned}$$

We need to check that the types are well-formed and the function well-typed. The side condition $\Gamma \vdash_\Delta$ ensures that the types are well-formed. To see that the function com^{+} is well-typed we consider the types of subterms.

- The term z has type $\forall\kappa.(A + B)$.

- The term $\pi_1\,(z[\kappa_0])$ has type $\mathbf{B}$.

- The term $\Lambda\kappa.\pi_2\,(z[\kappa])$ has type

$$\forall\kappa.\,\mathsf{El}\Big(\mathsf{if}\,\pi_1\,(z[\kappa])\,\mathsf{then}\,\widehat{A}\,\mathsf{else}\,\widehat{B}\Big)$$

- From TmEq-∀-fresh we get $\pi_1(z[\kappa_0]) \equiv \pi_1(z[\kappa])$. Indeed, the term

$$\Lambda\kappa.\pi_1(z[\kappa])$$

  has type $\mathbf{B}$, which does not contain κ, and the required equality follows from TmEq-∀-fresh and the β rule for clock quantification.

- Thus the term $\Lambda\kappa.\pi_2\,(z[\kappa])$ has type

$$\forall\kappa.\,\mathsf{El}\Big(\mathsf{if}\,\pi_1\,(z[\kappa_0])\,\mathsf{then}\,\widehat{A}\,\mathsf{else}\,\widehat{B}\Big)$$

- And so the term

$$\mathsf{com}^{\mathsf{if}}_{\pi_1(z[\kappa_0])}\,\Lambda\kappa.\pi_2\,(z[\kappa])$$

  has type

$$\mathsf{El}\Big(\mathsf{if}\,\pi_1\,(z[\kappa_0])\,\mathsf{then}\,\widehat{\forall}\,\Lambda\kappa.\widehat{A}\,\mathsf{else}\,\widehat{\forall}\,\Lambda\kappa.\widehat{B}\Big)$$

  which is exactly the type needed to typecheck the whole term.

For the term com^+ we can derive the following definitional term equalities.

$$\begin{aligned}\mathsf{com}^+\,(\Lambda\kappa.\,\mathsf{inl}\,t) &\equiv \mathsf{inl}\,\Lambda\kappa.t\\ \mathsf{com}^+\,(\Lambda\kappa.\,\mathsf{inr}\,t) &\equiv \mathsf{inr}\,\Lambda\kappa.t\end{aligned} \tag{3.12}$$

There is also a canonical term of type

$$\forall\kappa.A + \forall\kappa.B \to \forall\kappa.(A + B)$$

defined as

$$\begin{aligned}&\lambda z.\Lambda\kappa.\,\mathsf{case}\,z\,\mathsf{of}\\ &\qquad \mathsf{inl}\,a \Rightarrow \mathsf{inl}\,(a[\kappa])\\ &\qquad \mathsf{inl}\,b \Rightarrow \mathsf{inl}\,(b[\kappa]).\end{aligned}$$

This term is inverse to com^+, although we require equality reflection to show that the two functions are inverses to each other. Without equality reflection we can only prove they are inverses up to propositional equality. The isomorphisms defined previously do not require equality reflection.

Chapter 4

Cubical Types

This chapter consists of the paper:

[15] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi.
Guarded Cubical Type Theory: Path Equality for Guarded Recursion
In *Computer Science Logic (CSL)*, 2016.

along with a technical appendix.

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.

4.1 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 [13]. 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. [27], following earlier work [3, 8, 69], 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 [63] and Agda [70]. Bizjak et al. [22], following earlier work [14, 66], 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 [42, 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) [29] 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 [84]. 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 [29, 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. 4.2 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 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. 4.3 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,

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

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

[3] http://github.com/hansbugge/cubicaltt/tree/gcubical

but are included in a technical appendix[4].

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.

4.2.1 Cubical Type Theory

We first give a brief overview of *cubical type theory*[5] (CTT) [29]. 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 [62], 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.

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:

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

[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 4.1: Typing rules for path types.

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. 4.1. *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:

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

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

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.

4.2.2 Later Types

In Fig. 4.3 we present the 'later' types of guarded dependent type theory (GDTT) [22], with judgemental equalities in Figs. 4.4 and 4.5. 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

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

Figure 4.2: Formation rules for delayed substitutions.

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

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

Figure 4.3: Typing rules for later types.

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. 4.2. 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. [22] these constructions generalise the *applicative functor* [64] 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 [10] 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. 4.2.4.

Example 4.1. 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:

$$\mathsf{Id}_{\triangleright \xi.A}\, (\mathsf{next}\, \xi.\, t)\, (\mathsf{next}\, \xi.\, u) \;=\; \triangleright \xi.\, \mathsf{Id}_A\ t\ u \tag{4.1}$$

A term from left-to-right of (4.1) 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

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

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

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

Figure 4.4: Type equality rules for later types (congruence and equivalence rules are omitted).

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

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

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

$$\frac{\Gamma \vdash t : \triangleright\xi.A}{\Gamma \vdash \mathsf{next}\,\xi\,[x \leftarrow t].\,x = t : \triangleright\xi.A}$$

Figure 4.5: 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. 4.4.

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

Figure 4.6: Typing and equality rules for the delayed fixed-point

converting this into a path on later types. In fact in GCTT we can define a term with the desired type:

$$\lambda p.\langle i \rangle \mathsf{next}\, \xi[p' \leftarrow p].\, p'\, i \;:\; (\triangleright \xi.\, \mathsf{Path}\, A\, t\, u) \to \mathsf{Path}\, (\triangleright \xi.A)\, (\mathsf{next}\, \xi.\, t)\, (\mathsf{next}\, \xi.\, u). \tag{4.2}$$

Note the similarity of this term and type with that of funext, for functional extensionality, presented on page 103. Indeed we claim that (4.2) provides a computational interpretation of extensionality for later types.

4.2.3 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. 4.6. 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 4.2 (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 4.3 (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 (4.2). We compose s with p, and then with the inverse of the canonical unfold lemma of Lem. 4.2, 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

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

□

4.2.4 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. 4.2.2. 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

$$\triangleright\Big[y \leftarrow \mathsf{dfix}^0\, x.A \times \triangleright[y \leftarrow x].y\Big].y \tag{4.3}$$

rather than the desired $\triangleright\mathsf{Str}_A$. However we are not far off; $\triangleright\mathsf{Str}_A$ is judgementally equal to $\triangleright\Big[y \leftarrow \mathsf{dfix}^1\, x.A \times \triangleright[y \leftarrow x].y\Big].y$, which is the same term as (4.3), apart from endpoint 1 replacing 0. The canonical unfold lemma (Lem. 4.2) 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 4.4 (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. 4.2). 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:

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

The complete proof term, in the language of the type checker, can be found in Appendix 4.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 [13]. 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 $\triangleright$ [3]; what is new to GCTT is that we can also prove properties about it:

Proposition 4.5 (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. 4.3, 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 $f\,(\mathsf{next}(\mathsf{Y}\, f))$. □

4.3 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* [29] 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. [29], 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 [73, 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* [37] 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 $\widehat{\mathcal{C} \times \omega}$.

4.3.1 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 4.6.

- $\mathbb{F}$ *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 [29]. Given $\Gamma \vdash A$ we define the corresponding type of compositions as

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

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 [29]. 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 [60, 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 $\mathbb{D}$.

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

4.3.2 Adding Glueing and the Universe

The glueing construction [29, 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 [43] $\mathcal{U}^\omega$ 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. [29] 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.

[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 [81].

[7] This type is already present in Kapulkin at al. [53, Thm 3.4.1].

4.3.3 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 [22]. 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.

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 4.7. *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*

$$\Gamma \vdash \triangleright\xi.\Pi(x : A).B \cong \Pi(x : A).\triangleright\xi.B \tag{4.4}$$

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

Corollary 4.8. *If* $\Gamma \vdash \varphi : \mathbb{F}$ *then we have an isomorphism of types*

$$\Gamma \vdash \triangleright\xi.\Pi(p : [\varphi]).B \cong \Pi(x : [\varphi]).\triangleright\xi.B. \tag{4.5}$$

Lemma 4.9 ($\triangleright\xi$-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. 4.8 and Lem. 4.7 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:

$$\begin{aligned}
\triangleright\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\mathrm{Id}(a'_0, u'\, 0)^{\varphi} &\cong \left(\triangleright\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\mathrm{Id}(a'_0, u'\, 0)\right)^{\varphi} && \text{(Cor. 4.8)} \\
&= \left(\mathrm{Id}(\mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].a'_0, \mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].u'\, 0)\right)^{\varphi} \\
&= (\mathrm{Id}(a_0, u\, 0))^{\varphi},
\end{aligned}$$

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. 4.7). 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. 4.7). Assuming φ, we have

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

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.

4.4 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* [8], 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 [18]. 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) [5]. 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 [81] used the interval of the internal logic of cubical sets to model identity types. Coquand [32] 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 [71] axiomatises a model for CTT without a universe, again building on Coquand [32]. 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.

4.A zipWith Preserves Commutativity

We provide a formalisation of Sec. 4.2.4 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)
```

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

```
      (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])
```

4.B Guarded Cubical Type Theory

We define *guarded cubical type theory* (GCTT) to be an extension of cubical type theory (CTT)[9] [29] with the following syntax:

$$t, u, A, B \quad ::= \quad \cdots \mid \mathsf{next}\,\xi.\,t \mid \triangleright\xi.A \mid \mathsf{dfix}^r\, x.t$$

$$\xi \quad ::= \quad \cdot \mid \xi\,[x \leftarrow t]$$

along with the typing rules of Figure 4.7 and Figure 4.8.

4.C Denotational semantics

In this section we provide the necessary semantic constructions that can be used to interpret the type theory GCTT.

4.C.1 The language $\mathcal{L}$

Instead of formulating our model directly using regular mathematics, we will specify a type-theoretic language $\mathcal{L}$, tailor-made for the purpose of our model. It is based on the internal logic of the presheaf topos of cubical sets, $\mathrm{Set}^{\mathcal{C}}$.

$\mathcal{L}$ is an extension of W. Phoa's [73, Appendix I] *dependent predicate logic*; see also [51, D4.3,4.4]. Figure 4.9 contains an overview of the types of judgements. Note that a *proposition* is a term of type Ω. Formally the logical inference judgements will be of the form $\Gamma \vdash \varphi = \mathrm{true} : \Omega$, but in practice we will stick

[9] An overview of the rules of CTT can be found at http://www.cse.chalmers.se/~coquand/selfcontained.pdf.

Delayed substitutions, $\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$

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

Well-formed types, $\Gamma \vdash A$

$$\frac{\Gamma, \Gamma' \vdash A \qquad \vdash \xi : \Gamma \twoheadrightarrow \Gamma'}{\Gamma \vdash \triangleright \xi.A}$$

Well-typed terms, $\Gamma \vdash t : A$

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

$$\frac{\Gamma \vdash r : \mathbb{I} \qquad \Gamma, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{dfix}^r\, x.t : \triangleright A}$$

Figure 4.7: Overview of new rules in GCTT (part 1).

to a more informal notation, e.g., writing Γ, φ for a context where φ holds, instead of $\Gamma, p : \mathrm{Eq}(\varphi, \mathrm{true})$. In addition to the equality proposition $\mathrm{Eq}(t, u) : \Omega$, we also have an extensional identity type $\mathrm{Id}_A(t, u)$ with equality reflection:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t, u : A}{\Gamma \vdash \mathrm{Id}_A(t, y)} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash \mathrm{refl} : \mathrm{Id}_A(t, u)} \qquad \frac{\Gamma \vdash p : \mathrm{Id}_A(t, u)}{\Gamma \vdash t = u : A}$$

Id (the type) and Eq (the proposition) are equally expressive, but for presentation purposes it is practical to have both: Using Id we can easily express the type of *partial elements* without reference to Ω, e.g., an element of B only defined when $t = u$: $\Gamma \vdash b : \mathrm{Id}_A(t, u) \to B$. Such terms, however, are unwieldy to work with since you need to carry around an explicit equality proof (which will be equal to refl anyway). Therefore we will implicitly convert back and forth between the type theoretic and the logical representation, which for our previous example means that in a context where $t = u$ we will write $b : B$.

We also assume that $\mathcal{L}$ contains a universe $\mathcal{U}$ of small types, along with the "elements-of" functor El.

Assumption 1: The interval type

In $\mathcal{L}$ we have a type $\mathbb{I}$ with

$$0, 1 : \mathbb{I} \qquad \wedge, \vee : \mathbb{I} \to \mathbb{I} \to \mathbb{I} \qquad 1 - \cdot : \mathbb{I} \to \mathbb{I}$$

Type equality, $\Gamma \vdash A = B$ (Congruence and equivalence rules are omitted)

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

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

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

Term equality, $\Gamma \vdash t = u : A$ (Congruence and equivalence rules are omitted)

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

$$\frac{\begin{array}{c}\vdash \xi\,[x \leftarrow t, y \leftarrow u]\,\xi' : \Gamma \twoheadrightarrow \Gamma', x : B, y : C, \Gamma'' \\ \Gamma, \Gamma' \vdash C : \mathsf{U} \qquad \Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash A : \mathsf{U}\end{array}}{\Gamma \vdash \triangleright\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A = \triangleright\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A : \mathsf{U}}$$

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

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

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

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

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

Figure 4.8: Overview of new rules in GCTT (part 2).

$\Gamma \vdash$	well-formed context
$\Gamma \vdash A$	well-formed type
$\Gamma \vdash t : A$	typing judgement
$\Gamma \vdash A = B$	type equality
$\Gamma \vdash t = u : A$	term equality

Figure 4.9: Judgements of $\mathcal{L}$

which is a *De Morgan algebra*, i.e., $(\mathbb{I}, 0, 1, \wedge, \vee)$ is a bounded distributive lattice, and $1 - \cdot$ is an involution which satisfies De Morgan's laws:

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

In addition to the De Morgan algebra laws we assume the following two axioms

$$0 \neq 1$$
$$i \vee j = 1 \implies i = 1 \vee j = 1$$

the latter of which we refer to as the *disjunction property*.

Definable concepts

We can now define some useful abbreviations in $\mathcal{L}$.

Faces. Using the interval 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 in particular states that the inclusion is compatible with all the lattice operations, hence omitting it is justified.

Lemma 4.10.

- $\mathbb{F}$ *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.*
- $\mathbb{F}$ *inherits the disjunction property from* $\mathbb{I}$.

- *The operation* $\varphi = 1 \mapsto \varphi = 0$ *does* not *make* $\mathbb{F}$ *into a DM-algebra: For all* i, $(i = 1) \wedge ((1 - i) = 1) =_\Omega \bot$. *However, if* $((1 - i) = 1) \vee (i = 1) =_\Omega \top$, *then* $i = 0$ *or* $i = 1$.

Given a proposition $\Gamma \vdash \varphi : \mathbb{F}$ we define the type

$$[\varphi] \triangleq \mathrm{Id}_{\mathbb{F}}(\varphi, \top).$$

Remark 4.11. Note that we have the following logical equivalence

$$\Gamma \mid \cdot \vdash (\exists! p : [\varphi], \top) \iff \varphi.$$

Given $\Gamma \vdash A$ and $\Gamma \vdash \varphi : \mathbb{F}$ we say that a term t is a *partial element* of A of extent φ, if $\Gamma \vdash t : \Pi(p : [\varphi]).A$. 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. Given $\Gamma, p : [\varphi] \vdash B$ we write B^φ for the dependent function space $\Pi(p : [\varphi]).B$ and leave the proof p implicit.

If we have a term $\Gamma, p : [\varphi] \vdash u : A$ (a partial element), then we can define

$$A[\varphi \mapsto u] \triangleq \Sigma(a : A).(\mathrm{Id}_A(a, u))^\varphi.$$

Systems. Given $\Gamma \vdash A$, assume we have the following:

$$\begin{gathered}
\Gamma \vdash \varphi_1, \ldots, \varphi_n : \mathbb{F} \\
\Gamma \vdash \varphi_1 \vee \cdots \vee \varphi_n = \top \\
\Gamma, [\varphi_1] \vdash t_1 : A \\
\vdots \\
\Gamma, [\varphi_n] \vdash t_n : A \\
\Gamma, \left[\varphi_i \wedge \varphi_j\right] \vdash t_i = t_j : A, \quad \text{for all } i, j.
\end{gathered}$$

In other words: We have n partial elements of A which agree with each other. We can use the *axiom of definite description* to define a term

$$[\varphi_1 t_1, \ldots, \varphi_n t_n] \triangleq \text{the } x^A \text{ such that } \chi(x)$$

where

$$\chi(x) \triangleq (\varphi_1 \wedge (x = t_1)) \vee \cdots \vee (\varphi_n \wedge (x = t_n)).$$

We call this term a *system*. The condition for using definite description is a proof (in the logic) of *unique existence* of such a term. This follows almost directly from the assumptions and Remark 4.11.

Using systems, we generalise an earlier definition: We define

$$A[\varphi_1 \mapsto t_1, \ldots, \varphi_n \mapsto t_n] \triangleq A[\varphi_1 \vee \cdots \vee \varphi_n \mapsto [\varphi_1 t_1, \ldots, \varphi_n t_n]],$$

where the type on the right hand side is using the earlier definition. Note that $A[\varphi \mapsto t]$ is unambiguous, as we have $\Gamma, [\varphi] \vdash [\varphi t] = t : A$.

Compositions. Given $\Gamma \vdash A$, we can define the type of *compositions*:

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

We say that a type $\Gamma \vdash A$ is *fibrant* if there is a term $\vdash \mathbf{c} : \Phi(\Gamma; A)$ (*A has compositions*).

Fillings. Given $\Gamma \vdash A$, we can define the type of *(Kan) fillings*:

$$\begin{aligned}\Psi(\Gamma, A) \triangleq \Pi(&\gamma : \mathbb{I} \to \Gamma)\\ &(\varphi : \mathbb{F})\\ &(u : \Pi(i : \mathbb{I}).\, [\varphi] \to A(\gamma(i)))\\ &(a_0 : A(\gamma(0))[\varphi \mapsto u(0)])\\ &(i : \mathbb{I}).\\ &A(\gamma(i))[\varphi \mapsto u(i), (1 - i) \mapsto \pi_1 a_0].\end{aligned}$$

If we have a filling operation $\mathbf{f} : \Psi(\Gamma, A)$ then we can get a *path lifting* operation

$$\begin{aligned}\boldsymbol{\ell} : \Pi(&\gamma : \mathbb{I} \to \Gamma)\\ &(a_0 : A(\gamma(0)))\\ &(i : \mathbb{I}).\\ &A(\gamma(i))[(1 - i) \mapsto a_0],\end{aligned}$$

by taking the simple case of $\mathbf{f}$ where φ is $\bot$, and u therefore is uniquely determined (since it is a partial function defined where $\bot$ holds).

Fillings are special cases of compositions.

Lemma 4.12 (Fillings from compositions)**.** *If we have a fibrant type $\Gamma \vdash A$ with $\mathbf{c}_A : \Phi(\Gamma; A)$, then we have a filling operation $\vdash \mathbf{f} : \Psi(\Gamma, A)$.*

Proof. We introduce the variables of the proper types:

$$\begin{aligned}&\gamma : \mathbb{I} \to \Gamma,\\ &\phi : \mathbb{F},\\ &u : \Pi(i : \mathbb{I}).[\varphi] \to A(\gamma(i)),\\ &a_0 : A(\gamma(0))[\varphi \mapsto u(0)],\\ &i : \mathbb{I}.\end{aligned}$$

We need to find a term of type

$$A(\gamma(i))[\varphi \mapsto u(i), (i = 0) \mapsto \pi_1 a_o].$$

We check that the following system is well-defined (in a context with $\varphi \vee (i = 0)$):

$$[\varphi u(i \wedge j), (i = 0)\pi_1 a_0].$$

- If φ, then $u(i \wedge j) : A(\gamma(i \wedge j))$.
- If $i = 0$, then $\pi_1 a_0 : A(\gamma(0)) = A(\gamma(i \wedge j))$.
- If φ and $i = 0$, then $\pi_1 a_0 = u(0) = u(i \wedge j)$.

Note also that this means that

$$A(\gamma(0))[\varphi \mapsto u(0)] = A(\gamma(0))[\varphi \mapsto u(0), (i = 0) \mapsto \pi_1 a_0],$$

and therefore we can write the following term:

$$\mathbf{c}_A\ (\lambda j.\gamma(i \wedge j))\ (\varphi \vee (i = 0))\ (\lambda j.[\varphi u(i \wedge j), (i = 0)\pi_1 a_0])\ a_0$$

which has the type

$$A(\gamma(i))[\varphi \mapsto u(i), (i = 0) \mapsto \pi_1 a_o],$$

as was needed. □

Path types. Given $\Gamma \vdash A$ and terms $\Gamma \vdash t, u : A$, we can define the *Path type*

$$\mathrm{Path}_A\ t\ u \triangleq \Pi(i : \mathbb{I}).A[(1 - i) \mapsto t, i \mapsto u].$$

Assumption 2: Glueing

There is a type for *glueing* with the following type formation and typing rules

$$\frac{\Gamma \vdash A \qquad \Gamma, [\varphi] \vdash T \qquad \Gamma, [\varphi] \vdash f : T \to A}{\Gamma \vdash \mathrm{Glue}\ [\varphi \mapsto (T, f)]\ A} \qquad \frac{\Gamma \vdash b : \mathrm{Glue}\ [\varphi \mapsto (T, f)]\ A}{\Gamma \vdash \mathrm{unglue}\ b : A[\varphi \mapsto f\ b]}$$

$$\frac{\Gamma, [\varphi] \vdash f : T \to A \qquad \Gamma, [\varphi] \vdash t : T \qquad \Gamma \vdash a : A[\varphi \mapsto f\ t]}{\Gamma \vdash \mathrm{glue}\ [\varphi \mapsto t]\ a : \mathrm{Glue}\ [\varphi \mapsto (T, f)]\ A}$$

Additionally we have the following equations for glueing:

$$\begin{aligned} \mathrm{glue}\ [1 \mapsto t]\ a &= t, \\ \mathrm{glue}\ [\varphi \mapsto b]\ (\mathrm{unglue}\ b) &= b, \\ \mathrm{unglue}(\mathrm{glue}\ [\varphi \mapsto t]\ a) &= a. \end{aligned}$$

Assumption 3: Fibrant universe

There is a *fibrant universe* $\mathcal{U}_f$ which contains all of the codes in $\mathcal{U}$ for which there is an associated composition operator:

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

Assumption 4: $\forall$

Finally we assume that the map $\varphi \mapsto \lambda\_.\varphi : \mathbb{F} \to (\mathbb{I} \to \mathbb{F})$ has an internal right adjoint $\forall$. By this we mean the following correspondence for any $\varphi : \mathbb{F}$ and any $f : \mathbb{I} \to \mathbb{F}$.

$$(\varphi \Rightarrow \forall(i : \mathbb{I}), f(i)) \iff (\varphi \Rightarrow \forall(f)).$$

4.C.2 A model of CTT

We define a *category with families* [37] by specifying the type and term functors. The idea is to reuse the types and terms of the language $\mathcal{L}$, but we only take the *fibrant* types, i.e., the ones with associated composition operators.

$$\mathrm{Ty}(\Gamma) \triangleq \left\{ ([A], [\mathbf{c}_A]) \;\middle|\; \begin{array}{l} \Gamma \vdash A \\ \vdash \mathbf{c}_A : \Phi(\Gamma; A) \end{array} \right\}$$

$$\mathrm{Tm}(\Gamma, ([A], [\mathbf{c}_A])) \triangleq \{[t] \mid \Gamma \vdash t : A\}.$$

where we use $[A]$ and $[t]$ respectively for the equivalence classes of A and t modulo judgemental equality of $\mathcal{L}$. In the following we will omit the mention of equivalence classes and work with representatives. This is justified since all the operations in $\mathcal{L}$ respect judgemental equality.

Note, that the context Γ need not be fibrant. Context extension and projections can just be taken directly from the internal language: $\Gamma.A \triangleq \Sigma\Gamma A$, $\mathsf{p} \triangleq \pi_1, \mathsf{q} \triangleq \pi_2$.

When we interpret CTT we need to find both a type and a composition operator in $\mathcal{L}$ for each type in CTT.

Interpreting dependent function types

Assume that $[\![\Gamma \vdash A']\!] = (A, \mathbf{c}_A)$ and $[\![\Gamma, x : A' \vdash B']\!] = (B, \mathbf{c}_B)$. We define

$$[\![\Gamma \vdash (x : A') \to B']\!] \triangleq (\Pi(x : A).B, \mathbf{c})$$

where $\mathbf{c} : \Phi(\Gamma; \Pi(x : A).B)$ comes from the following lemma:

Lemma 4.13. *Π-types preserve compositions. I.e., if we have composition terms $\mathbf{c}_A : \Phi(\Gamma; A)$ and $\mathbf{c}_B : \Phi(\Gamma.A; B)$, then we can form a new composition $\mathbf{c}_{\Pi(x:A).B} : \Phi(\Gamma, \Pi(x : A).B)$.*

Proof. Recall that Π-types commutes with substitution:

$$(\Pi(x : A).B)(\gamma) = \Pi(x : A(\gamma)).B(\gamma),$$

where $B(\gamma)$ is a type in the context with A. We introduce the variables:

$$\begin{aligned}
&\gamma : \mathbb{I} \to \Gamma,\\
&\varphi : \mathbb{F},\\
&u : \Pi(i : \mathbb{I}).[\varphi] \to \Pi(a : A(\gamma(i))).B(\gamma(i)),\\
&c_0 : (\Pi(a : A(\gamma(0))).B(\gamma(0)))[\varphi \mapsto u(0)].
\end{aligned}$$

We need to find an element in

$$\Pi(a : A(\gamma(1))).B(\gamma(1)),$$

along with a proof that it is $u(1)$ when $\varphi = 1$.

Let $a_1 : A(\gamma(1))$ be given. We define $a(i) : A(\gamma(i))[i \mapsto a_1]$ by using path lifting on a_1, i.e.,

$$a(i) \triangleq \boldsymbol{\ell}\ (\lambda i.\gamma(1 - i))\ a_1\ (1 - i).$$

Then

$$b_1 \triangleq \mathbf{c}_B\ (\lambda i.\langle \gamma(i), a(i)\rangle)\ \varphi\ (\lambda i.u(i)(a(i)))$$

will have the type $B(\gamma(1))[\varphi \mapsto u(1)a_1]$. So $\lambda a_1.\pi_1 b_1$ has the type we are looking for. Now assume $\varphi = \top$; then $\lambda a_1.b_1 = \lambda a_1.u(i)a_1 = u(i)$, which is what we needed. □

The above proof is analogous to the equality judgement for compositions at Π-types in CTT [29].

Interpreting dependent sum types

Dependent sum types $(x : A) \times B$ are interpreted by Σ-types from $\mathcal{L}$, along with the composition operation that comes from the following lemma:

Lemma 4.14. *Σ-types preserve compositions. I.e., if we have composition terms $\mathbf{c}_A : \Phi(\Gamma; A)$ and $\mathbf{c}_B : \Phi(\Gamma.A; B)$, then we can form a new composition $\mathbf{c}_{\Sigma(x:A).B} : \Phi(\Gamma, \Sigma(x : A).B)$.*

This proof is analogous to the equality judgement for compositions at Σ-types in CTT [29].

Interpreting base types

If a type A is independent of $\mathbb{I}$, then we say it is *discrete*. Externally, this means that it is a constant presheaf, i.e., $A = \Delta(A')$ for some $A' \in \mathrm{Set}$, where $\Delta : \mathrm{Set} \to \mathrm{Set}^{\mathcal{C}}$ is the constant presheaf functor. Internally, it means that the following type is inhabited

$$\Pi(i, j : \mathbb{I})(a : \mathbb{I} \to A).a(i) = a(j).$$

Externally we have an isomorphism $(\Delta(A))^{\mathbb{I}} \cong \Delta(A)$, so if a type is discrete in the external sense, then it will also be discrete in the internal sense.

Lemma 4.15. *For any cubical set A and any $I \in \mathcal{C}$ and $i \notin I$ the function $\beta_I^i : A^{\mathbb{I}}(I) \to A(I, i)$ defined as*

$$\beta_I^i(f) = f_\iota(i),$$

where $\iota : I \to I, i$ is the inclusion, is an isomorphism. Moreover the family β is natural in I and i in the following sense. For any $J \in \mathcal{C}$ and $j \notin J$ and any $g : I \to J$ we have

$$A(g + (i \mapsto j)) \circ \beta_I^i = \beta_J^j \circ A^{\mathbb{I}}(g).$$

Corollary 4.16. *If the obvious morphism $A \to A^{\mathbb{I}}$ is an isomorphism, then A is isomorphic to an object of the form $\Delta(a)$ for some $a \in \mathrm{Set}$.*

Proof. The obvious morphism is of course the constant map $a \mapsto \lambda\_.a$. Using Lemma 4.15 we thus have that for each I and $i \notin I$, $A(\iota) : A(I) \to A(I, i)$ is an isomorphism, where, again, ι is the inclusion. From this we have that for all I, the inclusion $A(\iota_I) : A(\emptyset) \to A(I)$ is an isomorphism.

Define $a = A(\emptyset)$ and $\alpha : \Delta(a) \to A$ as

$$\alpha_I = A(\iota_I).$$

We then have for any $f : I \to J$ the following

$$A(f) \circ \alpha_I = A(f \circ \iota_I) = A(\iota_J).$$

The latter because $f \circ \iota_I$ and ι_J are both maps from the empty set, hence they are equal.

By the previous lemma each α_I is an isomorphism and by the preceding calculation α is a natural transformation. Hence α is a natural isomorphism. □

Lemma 4.17. *If A is isomorphic to $\Delta(a)$ for some $a \in \mathrm{Set}$ then the obvious morphism $A \to A^{\mathbb{I}}$ is an isomorphism.*

Proof. The inverse to the isomorphism β in Lemma 4.15 is the morphism α_I^i

$$\alpha_I^i(a)_f(j) = A([f, (i \mapsto j)])(a).$$

By assumption $A(\iota)$ for any inclusion $\iota : I \to I, i$ is an isomorphism. It is easy to compute that the canonical morphism $A \to A^{\mathbb{I}}$ arises as the composition of $A(\iota)$ and α_I^i. □

Proposition 4.18. *Let A be a cubical set. The formula*

$$i : \mathbb{I}, j : \mathbb{I}, f : (\mathbb{I} \to A) \mid \cdot \vdash f(i) = f(j)$$

holds in the internal language if and only if A is isomorphic to $\Delta(a)$ for some $a \in \mathrm{Set}$.

Proof. Suppose the formula holds. Then it is easy to see that the constant map from A to $A^{\mathbb{I}}$ is an isomorphism (the inverse is given, for instance, by evaluation at 0). Corollary 4.16 implies the result.

Conversely assume $A \cong \Delta(a)$ for some $a \in \mathrm{Set}$. Then by Lemma 4.17 the canonical map $\mathrm{const} : A \to A^{\mathbb{I}}$ is an isomorphism. Hence it is internally surjective. Thus for any $f : \mathbb{I} \to A$ there is an a in A, such that $\mathrm{const}\, a = f$. From this we immediately have $f(i) = f(j)$ for any i and j in $\mathbb{I}$. □

Lemma 4.19. *Every discrete type $\vdash A$ is fibrant, i.e., it has a composition operator $\mathbf{c}_A : \Phi(\cdot; A)$.*

Proof. Since A is discrete, we have that $u(0) = u(1)$ for any $u : \Pi(i : \mathbb{I}).[\varphi] \to A$. Therefore $A[\varphi \mapsto u(0)] = A[\varphi \mapsto u(1)]$, so we can choose the constant function $\lambda\gamma, \varphi, u, a.a$ to be $\mathbf{c}_A$, since this will be of type $\Phi(\cdot, A)$. □

If we have a composition operator $\mathbf{c}_A : \Phi(\cdot; A)$ then we can always construct a weakened version $\mathbf{c}'_A : \Phi(\Gamma; A)$ for any Γ, since A does not depend on Γ.

Therefore we can interpret the natural number type:

$$[\![\Gamma \vdash \mathsf{N}]\!] \triangleq (\mathbb{N}, \mathbf{c}_{\mathbb{N}}),$$

where $\mathbf{c}_{\mathbb{N}}$ is the composition that we get from Lemma 4.19.

Interpreting systems

We interpret the systems of CTT by using the systems of $\mathcal{L}$, and by using the fact that systems preserve compositions: If we have a system $\Gamma \vdash [\varphi_1 A_1, \ldots, \varphi_n A_n]$, then we can define a new composition using a system consisting of the compositions of all the components:

$$\begin{array}{rcl} \mathbf{c} & \triangleq & \lambda\gamma, \psi, u, a_0.[\varphi_1(\gamma\, i)(\mathbf{c}_{A_1}\, \gamma_1\, \psi\, u\, a_0), \ldots, \varphi_n(\gamma\, i)(\mathbf{c}_{A_n}\, \gamma_n\, \psi\, u\, a_0)] \\ & : & \Phi(\Gamma; [\varphi_1 A_1, \ldots, \varphi_n A_n]), \end{array}$$

where $\gamma_m : \mathbb{I} \to \Gamma, [\varphi_m]$ is the context map γ extended with the witness of $[\varphi_m]$.

Interpreting path types

We interpret the path types:

$$[\![\Gamma \vdash \mathrm{Path}\, A\ t\ s]\!] \triangleq (\mathrm{Path}_{A'}\, [\![t]\!]\ [\![s]\!], \mathbf{c}),$$

where $[\![A]\!] = (A', \mathbf{c}_A)$ and $\mathbf{c} : \Phi(\Gamma; \mathrm{Path}_{A'}\, [\![t]\!]\ [\![s]\!])$ comes from Lemma 4.20.

Lemma 4.20. *Path-types preserve composition, i.e., if $\Gamma \vdash A$ is fibrant, then for any $\Gamma \vdash t, s : A$, we will have a composition operator $\mathbf{c} : \Phi(\Gamma; \mathrm{Path}_A\ t\ s)$.*

Proof. First note that if we have $\Gamma \vdash \mathrm{Path}_A\ t\ s :$ and $\vdash \gamma : \Gamma$, then

$$(\mathrm{Path}_A\ t\ s)(\gamma) = \mathrm{Path}_{A(\gamma)}\ t(\gamma)\ s(\gamma) = \Pi(i : \mathbb{I}).A(\gamma)\begin{bmatrix} i = 0 \mapsto t(\gamma) \\ i = 1 \mapsto s(\gamma) \end{bmatrix}.$$

Now let

$$\begin{aligned} &\gamma : \mathbb{I} \to \Gamma \\ &\varphi : \mathbb{I} \\ &u : \Pi(j : \mathbb{I}).[\varphi] \to \mathrm{Path}_{A(\gamma j)}\ t(\gamma j)\ s(\gamma j) \\ &p_0 : (\mathrm{Path}_{A(\gamma 0)}\ t(\gamma 0)\ s(\gamma 0))[\varphi \mapsto u0] \end{aligned}$$

be given. Our goal is to find a term p_1 such that

$$p_1 : (\mathrm{Path}_{A(\gamma 1)}\ t(\gamma 1)\ s(\gamma 1))[\varphi \mapsto u1].$$

We will do this by finding a term $q : \Pi(i : \mathbb{I}).A(\gamma 1)[\varphi \mapsto u1i]$, for which we verify that $q0 = t(\gamma 1)$ and $q1 = s(\gamma 1)$, in other words,

$$q : \Pi(i : \mathbb{I}).A(\gamma 1)[\varphi \mapsto u1i, (1-i) \mapsto t(\gamma 1), i \mapsto s(\gamma 1)]$$

as this will be equivalent to having such a p_1.

Let $i : \mathbb{I}$. By leaving some equality proofs implicit we can define the system

$$r(j) \triangleq [\varphi uji, (1-i)t(\gamma j), is(\gamma j)] : \Pi(j : \mathbb{I}).[\varphi \vee (1-i) \vee i] \to A(\gamma j),$$

which is well-defined because $uj0 = t(\gamma j)$ and $uj1 = s(\gamma j)$. We also have that $p_0 i : A(\gamma 0)[\varphi \mapsto u0i]$, and since $p_0 0 = t(\gamma 0)$ and $p_0 1 = s(\gamma 0)$, we can say that

$$p_0 i : A(\gamma 0)[\varphi \mapsto u0i, (1-i) \mapsto t(\gamma 0), i \mapsto s(\gamma 0)]$$

so we can use the fibrancy of A to define the term

$$\begin{aligned} q(i) \quad &\triangleq \quad \mathbf{c}_A \gamma\ (\varphi \vee (1-i) \vee i)\ r\ (p_0 i) \\ &: \quad \Pi(i : \mathbb{I}).A(\gamma 1)[\varphi \mapsto u1i, (1-i) \mapsto t(\gamma 1), i \mapsto s(\gamma 1)], \end{aligned}$$

which is what we wanted. □

Interpreting glue types

We interpret Glue from CTT using Glue from $\mathcal{L}$ along with a composition operator, which we have by the following lemma:

Lemma 4.21. *Glueing is fibrant, i.e., if we have*

$$\begin{aligned} \Gamma &\vdash A \\ \Gamma &\vdash \varphi : \mathbb{I} \\ \Gamma, [\varphi] &\vdash T \\ \Gamma &\vdash w : [\varphi] \to T \to A \\ \Gamma &\vdash p : \mathrm{isEquiv}\, w \end{aligned}$$

then there is a term $\mathbf{c} : \Phi(\Gamma; \mathrm{Glue}\, [\varphi \mapsto (T, w)]\, A)$.

The construction of $\mathbf{c}$ in the proof of the above lemma is analogous to the construction of the composition operation for glueing in CTT [29], but formulated in $\mathcal{L}$. A crucial part of the construction is the face $\delta \triangleq \forall(\varphi \circ \gamma)$, where $\gamma : \mathbb{I} \to \Gamma$, which satisfies that $[\delta]$ implies $[\varphi(\gamma\, i)]$ for all $i : \mathbb{I}$.

Interpreting the universe

The universe of CTT is interpreted using the universe of fibrant types $\mathcal{U}_f$. To define the composition for the universe we follow the construction of Cohen et al. [29] in the language $\mathcal{L}$.

4.C.3 A concrete model of $\mathcal{L}$

Given a countable set of names let Fin be the full subcategory of Set on finite subsets of names. Let $\mathcal{C}$ be the *opposite* of the Kleisli category of the free De Morgan algebra monad on Fin. The category of *cubical sets* is the presheaf category $\widehat{\mathcal{C}}$.

It is well-known how to model dependent predicate logic in any presheaf topos, so we omit the verification of this part. We do however note how the judgements are interpreted since this will be used later on in calculations.

- A context $\Gamma \vdash$ is interpreted as a presheaf.
- The judgement $\Gamma \vdash A$ gives a pair of a presheaf Γ on $\mathcal{C}$ and a presheaf A on the category of elements of Γ.
- The judgement $\Gamma \vdash t : A$ in addition gives a global element of the presheaf A. Thus for each $I \in \mathcal{C}$ and $\gamma \in \Gamma(I)$ we have $t_{I,\gamma} \in A(I, \gamma)$.

Moreover, there is a canonical bijective correspondence between presheaves Γ on $\mathcal{C}$ and interpretations of types $\cdot \vdash \Gamma$. This justifies treating contexts as types in $\mathcal{L}$ when it is convenient to do so.

Assumption 1 is satisfied

Take $\mathbb{I}$ to be the functor mapping $I \mapsto \hom[\mathcal{C}]I1$, where 1 is the (globally) chosen singleton set. Since the theory of De Morgan algebras is geometric and for each I we have a De Morgan algebra together with the fact that the morphisms are De Morgan algebra morphisms, we have that $\mathbb{I}$ is an internal De Morgan algebra, as needed.

Moreover the disjunction property axiom is also geometric, and since it is clearly satisfied by each free De Morgan algebra $\mathbf{DM}(I)$, it also holds internally.

Finally, it is easy to check that we have $0 = 1 \Rightarrow \bot$ using Kripke-Joyal semantics.

Assumption 2 is satisfied

We will define glueing almost internally, apart from a "strictness" fix, for which we use the following lemma, which we will also use later on in Section 4.C.5

A strictification lemma

Lemma 4.22. *Let C be a small category and $\top$ a global element*[10] *of an object $\mathbb{K}$ in $\widehat{C}$. Denote by $[\varphi]$ the identity type $\varphi = \top$.*

Let $\Gamma \vdash \varphi : \mathbb{K}$. Suppose $\Gamma \vdash T$, $\Gamma, [\varphi] \vdash A$ and $\Gamma, [\varphi] \vdash T \cong A$ as witnessed by the terms α, β satisfying

$$\Gamma, [\varphi], x : A \vdash \alpha : T$$
$$\Gamma, [\varphi], x : T \vdash \beta : A$$

plus the equations stating that they are inverses.

Then there exists a type $\Gamma \vdash \mathcal{T}(A, T, \varphi)$ such that

1. $\Gamma, [\varphi] \vdash \mathcal{T}(A, T, \varphi) = A$
2. $\Gamma \vdash T \cong \mathcal{T}(A, T, \varphi)$ *by an isomorphism α', β' extending α and β. This means that the following two judgements hold.*

   $$\Gamma, [\varphi], x : A \vdash \alpha = \alpha' : T$$
   $$\Gamma, [\varphi], x : T \vdash \beta = \beta' : A.$$

   The judgements are well-formed because in context $\Gamma, [\varphi]$ the types $\mathcal{T}(A, T, \varphi)$ and A are equal by the first item of this lemma.
3. *Let $\rho : \Delta \to \Gamma$ be a context morphism. Consider its extension $\Delta, [\varphi\rho] \to \Gamma, [\varphi]$. Then $\mathcal{T}(A, T, \varphi)\rho = \mathcal{T}(A\rho, T\rho, \varphi\rho)$.*

[10] For a constructive meta-theory we add that, for each c, equality with $\top_c$ is decidable.

Proof. We write T' for $\mathcal{T}(A, T, \varphi)$ and define it as follows.

$$T'(c,\gamma) = \begin{cases} A(c,(\gamma,\star)) & \text{if } \varphi_{c,\gamma} = \top_c \\ T(c,\gamma) & \text{otherwise} \end{cases}$$

Here $\star$ is the unique proof of $[\varphi]$. The restrictions are important. Given $f : (c,\Gamma(f)(\gamma)) \to (d,\gamma)$ define $T'(f)$ by cases

$$T'(f)(x) = \begin{cases} A(f)(x) & \text{if } \varphi_d(\gamma) = \top_d(\star) \\ \beta_{c,\Gamma(f)(\gamma),\star,T(f)(x)} & \text{if } \varphi_{c,\Gamma(f)(\gamma)} = \top_c \\ T(f)(x) & \text{otherwise} \end{cases}$$

We need to check that this definition is functorial. The fact that $T'(id) = id$ is trivial. Given $f : (d,\Gamma(f)(\gamma)) \to (c,\gamma)$ and $g : (e,\Gamma(f \circ g)(\gamma)) \to (d,\Gamma(f)(\gamma))$ we have

$$T'(f \circ g)(x) = \begin{cases} A(f \circ g)(x) & \text{if } \varphi_{c,\gamma} = \top_c \\ \beta_{e,\Gamma(f\circ g)(\gamma),\star,T(f\circ g)(x)} & \text{if } \varphi_{e,\Gamma(f\circ g)(\gamma)} = \top_e \\ T(f \circ g)(x) & \text{otherwise} \end{cases}$$

In the first and third cases this is easily seen to be the same as $T'(g)(T'(f)(x))$, since if $\varphi_{e,\Gamma(f\circ g)(\gamma)} \neq \top_e$ then also $\varphi_{d,\Gamma(f)(\gamma)} \neq \top_d$ by naturality of φ and the fact that $\top$ is a global element and the terminal object is a constant presheaf.

So assume the remaining option is the case, that is, $\varphi_{e,\Gamma(f\circ g)(\gamma)} = \top_e$ but $\varphi_{c,\gamma} \neq \top_c$.

We split into two further cases.

- Case $\varphi_{d,\Gamma(f)(\gamma)} = \top_d$. Then $T'(f)(x) = \beta_{d,\Gamma(f)(\gamma),\star,T(f)(x)}$ and so

  $$T'(g)(T'(f)(x)) = T'(g)\left(\beta_{d,(\Gamma(f)(\gamma),\star,T(f)(x))}\right)$$

  By naturality of β the right-hand side is the same as

  $$\beta_{e,\Gamma(f\circ g)(\gamma),\star,T(f\circ g)(x)}$$

  which is what is needed.

- Case $\varphi_{d,\Gamma(f)(\gamma)} \neq \top_d$. In this case we have

  $$T'(f)(x) = T(f)(x)$$

  and

  $$T'(g)(T'(f)(x)) = \beta_{e,\Gamma(f\circ g)(\gamma),\star,T(g)(T(f)(x))}$$

  which is again, as needed by functoriality of T.

Now, directly from the definition we have the equality $\Gamma, [\varphi] \vdash T' = A$.

It is similarly easy to check the last required property, the naturality of the construction.

$$T(A, T, \varphi)\rho = T(A\rho, T\rho, \varphi\rho).$$

Finally, we extend the isomorphisms α and β to α' and β'.

Define β' satisfying $\Gamma, x : T \vdash \beta' : T'$ as

$$\beta'_{c,\gamma,x} = \begin{cases} \beta_{c,\gamma,\star,x} & \text{if } \varphi_c(\gamma) = \top_c(\star) \\ x & \text{otherwise} \end{cases}$$

And α' analogously. One needs to check that this is a natural transformation, i.e., a global element. Finally, β' is the inverse to α' by construction. □

Given the following types and terms

$$\begin{aligned} \Gamma &\vdash \varphi : \mathbb{F} \\ \Gamma, [\varphi] &\vdash T \\ \Gamma &\vdash A \\ \Gamma, [\varphi] &\vdash w : T \to A \end{aligned}$$

we define a new type $\Gamma \vdash \mathrm{Glue}\,[\varphi \mapsto (T, w)]\, A$ in two steps.

First we define the type $Glue'_\Gamma(\varphi, T, A, w)$ in context Γ as

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

For this type we have the following property (we write G' for $Glue'(\cdots)$)

$$\Gamma, [\varphi] \vdash T \cong G'$$

with the isomorphism consisting of the second projection from right to left and from left to right we use w to construct the pair.

Finally, we define $\mathrm{Glue}\,[\varphi \mapsto (T, w)]\, A$ using Lemma 4.22 applied to the type $Glue'$. Let

$$\beta : \mathrm{Glue}\,[\varphi \mapsto (T, w)]\, A \to Glue'(\varphi, T, A, w)$$

be the extension of pairing and

$$\alpha : Glue'(\varphi, T, A, w) \to \mathrm{Glue}\,[\varphi \mapsto (T, w)]\, A$$

the extension of the projection as per Lemma 4.22.

Define $\mathrm{unglue} : \mathrm{Glue}\,[\varphi \mapsto (T, w)]\, A \to A$ be the composition of β and the *first* projection $G' \to A$. Now if $\varphi = \top$ then β is just pairing and in this case we also have $\mathrm{Glue}\,[\varphi \mapsto (T, w)]\, A = T$. So by definition of G' we have $\mathrm{unglue}(t) = wt$, validating one of the equalities.

Given $\Gamma, [\varphi] \vdash t : T$ and $\Gamma \vdash a : A$ satisfying $a = wt$ on $[\varphi]$ define $\Gamma \vdash \mathrm{glue}\,[\varphi \mapsto t]\, a : \mathrm{Glue}\,[\varphi \mapsto (T, w)]\, A$ to be pairing followed by α. If $\varphi = \top$ we have, because α is just the projection in this case, that $\mathrm{glue}\,[1 \mapsto t]\, a = t$.

Assumption 3 is satisfied

This part is subsumed by the construction in Section 4.C.4.

Assumption 4 is satisfied

Theorem 4.23. *$\widehat{\mathcal{C}}$ models an operation $\forall : \mathbb{F}^{\mathbb{I}} \to \mathbb{F}$ which is right-adjoint to the constant map of posets $\mathbb{F} \to \mathbb{F}^{\mathbb{I}}$.*

Proof. We will first give a concrete description of $\mathbb{I}$ and $\mathbb{F}$. We know that $\mathbb{I}(n) = DM(n)$. We use Birkhoff duality [17] between finite distributive lattices and finite posets. This duality is given by a functor $J = \mathbf{Hom}_{\mathrm{fDL}}(-, \mathbb{2})$ from finite distributive lattices to the opposite of the category of finite posets. This functor sends a distributive lattice to its join-irreducible elements. It's inverse is the functor $\mathbf{Hom}_{\mathrm{poset}}(-, \mathbb{2})$ which sends a poset to its the distributive lattice of lower sets. This restricts to a duality between free distributive lattices and powers of $\mathbb{2}$. Every free *De Morgan* algebra on n generators is a free distributive lattice on $2n$ generators. We obtain a duality with the category of *even* powers of $\mathbb{2}$ and maps preserving the De Morgan involution [33]. Moreover, this duality is poset enriched: If $\psi \leq \varphi : DM(n) \to DM(m)$, then the corresponding maps on even powers of $\mathbb{2}$, which are defined by pre-composition, are in the same order relation.

The dual of the inclusion map is the projection $p : \mathbb{2}^{2(n+1)} \to \mathbb{2}^{2n}$. This has a right adjoint: concatenation with 11: $p\alpha \leq \beta$ iff $\alpha \leq \beta \cdot 11$. Concatenation with 11 is natural:

$$\begin{array}{ccc} \mathbb{2}^{2n} & \xrightarrow{f} & \mathbb{2}^{2m} \\ \uparrow\downarrow 11 & & \uparrow\downarrow 11 \\ \mathbb{2}^{2(n+1)} & \xrightarrow{(f,id)} & \mathbb{2}^{2(m+1)} \end{array}$$

By duality we obtain a natural right adjoint to the poset-inclusion of DM-algebras. Finally, we recall that in $\widehat{\mathcal{C}}$ we have $\mathbb{I}^{\mathbb{I}}(n) = \mathbb{I}(n+1)$ and hence we have an internal map $\forall : \mathbb{I}^{\mathbb{I}} \to \mathbb{I}$ which is right-adjoint to the constant map $\mathbb{I} \to \mathbb{I}^{\mathbb{I}}$.

In [29] the face lattice is defined as the quotient of $\mathbb{I}$ by the congruence $x \wedge 1 - x = 0$, for all x. In cubical sets these two definitions coincide. Let us temporarily write $\mathbb{F}_q$ for the image of $\mathbb{I}$ in Ω. Since $\mathbb{F}$ satisfies $[i] \wedge [1 - i] = \bot$, because $0 \neq 1$. So, we have a surjective lattice map: $\mathbb{F}_q \to \mathbb{F}$. We will show that it is also injective. Let φ, ψ be (generalised) elements of $\mathbb{I}$. Suppose that $[\psi] = [\varphi]$, i.e. $\psi = 1$ iff $\varphi = 1$. Then the sieves of functions in $\mathcal{C}$ which evaluate ψ, φ to 1 are the same. So, we may assume that ψ, φ have the same free variables. By considering the disjunctive normal forms of φ, ψ we see they should have the same conjuncts upto the equality $x \wedge -x = 0$. Injectivity follows. Hence, $\mathbb{F}_q = \mathbb{F}$.
The quotient presentation has the advantage of having decidable equality (externally) in the model, and hence this is used in the implementation. It is

also geometric since the construction of a distributive lattice by generators (from $\mathbb{I}$) and relations can be done by considering a quotient of $\mathcal{F}\mathcal{F}(\mathbb{I})$, where $\mathcal{F}$ denote the (Kuratowski) finite power set. Both quotients and $\mathcal{F}$ which are geometric. This will be useful later.

As just explained $\mathbb{F}$ is the quotient of $\mathbb{I}$ by the relation $x \wedge 1 - x = 0$, for all x. This is a geometric formula and hence holds at each stage n. As duality turns the quotients into inclusions, we have the inclusion $\{01, 10, 11\}^n \subset 2^{2n}$ as the set of join irreducible elements; as 00 presents $x \wedge -x$ which is now identified with $\bot$ and hence no longer join-irreducible. This presentation allows us to define $\forall : \mathbb{F}^{\mathbb{I}} \to \mathbb{F}$. Since $\mathbb{F}^{\mathbb{I}}(n) = \mathbb{F}(n+1)$, so the right adjoint is again given by concatenation by 11. We just replace 2^2 by $\{01, 10, 11\}$ in the diagram above. □

4.C.4 More models of $\mathcal{L}$

Lemma 4.24. *Let $\mathbb{C}, \mathbb{D}$ be small categories and let $\pi : \mathbb{D} \times \mathbb{C} \to \mathbb{D}$ be the projection functor. Then the geometric morphism $\pi^* \dashv \pi_*$ is open. If $\mathbb{C}$ is inhabited then it is also surjective.*

Proof. By Theorem C.3.1.7 of [51] it suffices to show that π^* is sub-logical. We use Lemma C.3.1.2 of loc. cit. to show this (we use the notation introduced in that lemma).

Let $b : \pi(I, n) \to J$ be a morphism in $\mathbb{D}$. Let $U' = (J, n)$, $a = (b, id_n) : (I, n) \to (J, n)$, $r = id_J : \pi U' \to J$ and $i = id_J : J \to \pi U'$. Then we have $r \circ i = id_J$ and $i \circ b = \pi a$ as required by Lemma C.3.1.2.

If $\mathbb{C}$ is inhabited the projection π is surjective on objects, so the corresponding geometric morphism is surjective; see [51, A4.2.7b] □

The previous lemma may be read as $\widehat{\mathbb{D} \times \mathbb{C}}$ is a conservative extension of $\hat{\mathbb{D}}$, provided $\mathbb{C}$ is inhabited.

Lemma 4.25. *If $\mathbb{C}$ has an initial object 0, then π^* is full, faithful, and cartesian closed.*

Proof. The functor π has a *left* adjoint, which is the functor

$$\iota : \mathbb{D} \to \mathbb{D} \times \mathbb{C}$$
$$\iota(I) = (I, 0)$$

Trivially we have $\pi \circ \iota = id_{\mathbb{D}}$. Thus we have that ι^* is *left* adjoint to π^* and because $\pi \circ \iota = id_{\mathbb{D}}$ we also have $\iota^* \circ \pi^* = id$ and moreover the counit of the adjunction is the identity. Hence the functor π^* is full and faithful [59, Theorem IV.3.1] and by [51, Corollary A.1.5.9], since ι^* preserves all limits, we have that π^* cartesian closed. □

Let $\Omega^{\mathbb{C}}$ be the subobject classifier of $\widehat{\mathcal{C} \times \mathbb{C}}$ and $\Omega^{\mathcal{C}}$ the subobject classifier of $\widehat{\mathcal{C}}$.

Lemma 4.26. *There is a* monomorphism $v : \pi^*\left(\Omega^{\mathcal{C}}\right) \to \Omega^{\mathbb{C}}$ *which fits into the pullback*

$$\begin{array}{ccc} \pi^*(1) & \xrightarrow{\cong} & 1 \\ {\scriptstyle \pi^*(\mathrm{true})}\downarrow & \lrcorner & \downarrow{\scriptstyle \mathrm{true}} \\ \pi^*\left(\Omega^{\mathcal{C}}\right) & \xrightarrow{v} & \Omega^{\mathbb{C}} \end{array}$$

Proof. Since π^* preserves monos $\pi^*(\mathrm{true})$ is a mono, hence define v to be its characteristic map. Concretely it maps

$$v_{I,c}(S) = \{(f, g) \mid f \in S\}$$

so it is clearly a mono. □

Corollary 4.27. *If $X = \pi^*(Y)$ then the equality predicate $\chi_\delta : X \times X \to \Omega^{\mathbb{C}}$ factors uniquely through v and the inclusion of the equality predicate of Y.*

Proof. The equality predicate is by definition the characteristic map of the diagonal $\delta : X \to X \times X$. Let $\delta' : Y \to Y \times Y$ be the diagonal. Because π^* preserves finite limits the following square is a pullback.

$$\begin{array}{ccccc} X & \longrightarrow & \pi^*(1) & \xrightarrow{\cong} & 1 \\ {\scriptstyle \delta = \pi^*(\delta')}\downarrow & \lrcorner & \downarrow{\scriptstyle \pi^*(\mathrm{true})} & \lrcorner & \downarrow{\scriptstyle \mathrm{true}} \\ X \times X & \xrightarrow{\pi^*(\chi_{\delta'})} & \pi^*\left(\Omega^{\mathcal{C}}\right) & \xrightarrow{v} & \Omega^{\mathbb{C}} \end{array}$$

and by uniqueness of characteristic maps we have $v \circ \pi^*(\chi_{\delta'}) = \chi_\delta$. Uniqueness of the factorisation follows from the fact that v is a mono. □

Let $\mathbb{C}$ be a category with an initial object. We now show that $\widehat{\mathcal{C} \times \mathbb{C}}$ models $\mathcal{L}$.

Assumption 1 is satisfied

Let $\mathbb{I}^{\mathbb{C}} = \pi^*(\mathbb{I})$. Since π^* preserves products we can lift all the De Morgan algebra operations of $\mathbb{I}$ to operations on $\mathbb{I}^{\mathbb{C}}$. The theory of a De Morgan algebra with a disjunction property and $0 \neq 1$ is geometric [60, Section X.3]. Thus the geometric morphism $\pi^* \dashv \pi_*$ preserves validity of all the axioms, which means that $\mathbb{I}^{\mathbb{C}}$ is an internal De Morgan algebra with $0 \neq 1$ and the disjunction property.

Faces

Lemma 4.28. *Let $\mathbb{F}^{\mathbb{C}} \in \widehat{\mathcal{C} \times \mathbb{C}}$ and $\mathbb{F} \in \widehat{\mathcal{C}}$ be defined as in Section 4.C.1 from $\mathbb{I}^{\mathbb{C}}$ and $\mathbb{I}$. Then $\mathbb{F}^{\mathbb{C}} \cong \pi^*(\mathbb{F})$.*

Proof. Let $e : \mathbb{I}^{\mathbb{C}} \to \Omega^{\mathbb{C}}$ be the composition $\chi_\delta \circ \langle id, 1 \rangle$ where δ is the diagonal $\mathbb{I}^{\mathbb{C}} \to \mathbb{I}^{\mathbb{C}} \times \mathbb{I}^{\mathbb{C}}$. By definition $\mathbb{F}^{\mathbb{C}}$ is the image of e. By Corollary 4.27 and the way we have defined $\mathbb{I}^{\mathbb{C}}$ and all the operations on it we have that $e = v \circ \pi^*(e')$ where $e' : \mathbb{I} \to \Omega^{\mathbb{C}}$ is defined analogously to e above.

By definition $\mathbb{F}$ is the image of e'. Because inverse images of geometric morphisms preserve image factorisations $\pi^*(\mathbb{F})$ is the image of $\pi^*(e')$. Finally, because v is a mono the image of $v \circ \pi^*(e')$ is canonically isomorphic to the image of $\pi^*(e')$, which is what the lemma claims. □

Assumption 2 is satisfied

This proceeds exactly as in Section 4.C.3.

Assumption 3 is satisfied

Lemma 4.29. *Let $\mathbb{C}$ and $\mathbb{D}$ be small categories and assume $\mathbb{D}$ has products. Let $k_1 : \mathbb{D} \to \widehat{\mathbb{D}}$ and $k_2 : \mathbb{C} \times \mathbb{D} \to \widehat{\mathbb{C} \times \mathbb{D}}$ be the Yoneda embeddings. Let $\pi^* : \widehat{\mathbb{D}} \to \widehat{\mathbb{C} \times \mathbb{D}}$ be the constant presheaf functor.*

For any $d, e \in \mathbb{D}$ and any $c \in \mathbb{C}$ there is an isomorphism

$$k_2(c,d) \times \pi^*(k_1 e) \cong k_2(c, d \times e)$$

in $\widehat{\mathbb{C} \times \mathbb{D}}$ which is natural in c, d and e.

Proof. For any $(c', d') \in \mathbb{C} \times \mathbb{D}$

$$\begin{aligned}(k_2(c,d) \times \pi^*(k_1 e))(c',d') &= \hom[\mathbb{C} \times \mathbb{D}](c',d')(c,d) \times \hom[\mathbb{D}]d'e \\ &\cong \hom[\mathbb{D}]d'd \times \hom[\mathbb{C}]c'c \times \hom[\mathbb{D}]d'e\end{aligned}$$

and because the hom functor preserves products we have

$$\begin{aligned}&\cong \hom[\mathbb{D}]d'd \times e \times \hom[\mathbb{C}]c'c \\ &\cong \hom[\mathbb{C} \times \mathbb{D}](c',d')(c, d \times e) \\ &= k_2(c, d \times e)(c', d')\end{aligned}$$

as required. □

The following lemma is useful for describing the composition externally, which is needed do define the fibrant universe.

Lemma 4.30. *Let $\mathbb{C}$ be a small category. Let $\mathbb{I}^{\mathbb{C}} \in \widehat{\mathbb{C} \times \mathcal{C}}$ be the inclusion $\pi^*(\mathbb{I})$ of $\mathbb{I} \in \widehat{\mathcal{C}}$. Let $X \in \widehat{\mathcal{C} \times \mathbb{C}}$. Then for any $c \in \mathbb{C}$, any $I \in \mathcal{C}$ and any $i \notin I$ we have*

$$X^{\mathbb{I}^{\mathbb{C}}}(I, c) \cong X((I, i), c)$$

naturally in c, I and i.

Remark 4.31. Note that disjoint union is the *coproduct* in the Kleisli category of the free De Morgan algebra monad. Hence disjoint union is the product in $\mathcal{C}$.

Proof. Using the Yoneda lemma and the defining property of exponents we have

$$\begin{aligned} X^{\mathbb{I}^{\mathbb{C}}}(I, c) &\cong \hom[\widehat{\mathcal{C} \times \mathbb{C}}]y(I, c)X^{\mathbb{I}^{\mathbb{C}}} \\ &\cong \hom[\widehat{\mathcal{C} \times \mathbb{C}}]y(I, c) \times \pi^*(\mathbb{I})X \end{aligned}$$

which by Lemma 4.29, together with the fact that $\mathbb{I}$ is isomorphic to $y\{i\}$, is isomorphic to

$$\begin{aligned} &\cong \hom[\widehat{\mathcal{C} \times \mathbb{C}}]y(I \cup \{i\}, c)X \\ &\cong X(I \cup \{i\}, c). \end{aligned}$$

Concretely, the isomorphism $\alpha^c_{I,i}$ maps $\xi \in X^{\mathbb{I}^{\mathbb{C}}}(I, c)$ to $\xi_{(\iota_{I,i}, id_c,)}(i)$, where $\iota_{I,i} : I \to I, i$ (in $\mathcal{C}^{\mathrm{op}}$) is the inclusion. Its inverse $\beta^c_{I,i}$ maps $x \in X(I \cup \{i\}, c)$ to the family of functions $\xi_{(f,g)} : \mathbb{I}(J) \to X(J, d)$ indexed by morphisms $(f, g) : (J, d) \to (I, c)$ (in $(\mathcal{C} \times \mathbb{C})$). This family is defined as

$$\xi_{(f,g)}(\varphi) = X([g, i \mapsto \varphi], g)(x)$$

where $[g, i \mapsto \varphi]$ is the map $I, i \to J$ (in $\mathcal{C}^{\mathrm{op}}$) which maps i to φ and otherwise acts as g. This map is well-defined because disjoint union is the coproduct in $\mathcal{C}^{\mathrm{op}}$. □

We can now define the universe $\mathcal{U}^{\mathbb{C}}_f$. First, the Hofmann-Streicher universe $\mathcal{U}^{\mathbb{C}}$ in $\widehat{\mathcal{C} \times \mathbb{C}}$ maps (I, c) to the set of functors valued in $\mathfrak{U}$ on the category of elements of $y(I, c)$. It acts on morphisms $(I, c) \to (J, d)$ by composition (in the same way as substitution in types is modelled).

The elements map

$$\frac{\Gamma \vdash a : \mathcal{U}^{\mathbb{C}}}{\Gamma \vdash \mathsf{El}(a)}$$

is interpreted as

$$\mathsf{El}(a)((I, c), \gamma) = a_{(I,c),\gamma}(\star)\,(id_{I,c}),$$

recalling that terms are interpreted as global elements, and $\star$ is the unique inhabitant of the chosen singleton set.

We define $\mathcal{U}_f^{\mathbb{C}}$ analogously to the way it is defined in Section 4.C.3, that is

$$\mathcal{U}_f^{\mathbb{C}}(I,c) = \mathrm{Ty}(y(I,c)).$$

We first look at the following rule.

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

Let us write $b = (\!|a, \mathbf{c}|\!)$. We need to give for each $I \in \mathcal{C}$, $c \in \mathbb{C}$ and $\gamma \in \Gamma(I,c)$ a pair (b_0, b_1) where

$$y(I,c) \vdash b_0 : \mathcal{U}^{\mathbb{C}}$$
$$\cdot \vdash b_1 : \Phi(y(I,c); \mathrm{El}(b_0))$$

Now b_0 is easy. It is simply $a_{(I,c),\gamma}$. Composition is also conceptually simple, but somewhat difficult to write down precisely. Elements $\gamma \in \Gamma(I,c)$ are in bijective correspondence (by Yoneda and exponential transpose) to terms $\overline{\gamma}$

$$\cdot \vdash \overline{\gamma} : y(I,c) \to \Gamma.$$

Thus we define

$$b_1 = \lambda\rho.\mathbf{c}\,(\overline{\gamma} \circ \rho).$$

One checks that this is well-defined and natural by a tedious computation, which we omit here.

We now look at the converse rule in $\mathcal{L}$

$$\frac{\Gamma \vdash a : \mathcal{U}_f}{\Gamma \vdash \mathrm{El}(a)} \qquad \frac{\Gamma \vdash a : \mathcal{U}_f}{\vdash \mathrm{Comp}(a) : \Phi(\Gamma; \mathrm{El}(a))}.$$

To interpret this rule with $\mathcal{U}_f^{\mathbb{C}}$, we interpret for any a and $\mathbf{c}$, $\mathrm{El}((\!|a, \mathbf{c}|\!))$ by $\mathrm{El}(a)$, where the latter is El map of the Hofmann-Streicher universe.

We need to define $\mathrm{Comp}(a)$ which we abbreviate to c. We need to give for each $I \in \mathcal{C}$ and $c \in \mathbb{C}$ an element $c_{I,c} \in \Phi(\Gamma; \mathrm{El}(a))(I,c)$, and this family needs to be natural in I and c. Given $\gamma \in (\Gamma^{\mathbb{I}^{\mathbb{C}}})(I,c)$ and a fresh $i \notin I$ we get by Lemma 4.30 an element $\gamma' \in \Gamma((I,i),c)$. Let $\overline{\gamma'} : y((I,i),c) \to \Gamma$ be the morphism corresponding to γ' by the Yoneda lemma. Thus we get from a the term $c'_{I,i,c,\gamma}$

$$\cdot \vdash c'_{I,i,c,\gamma} : \Phi(y((I,i),c); \mathrm{El}(a)\overline{\gamma'})$$

and hence by weakening a term

$$y(I,c) \vdash c'_{I,i,c,\gamma} : \Phi(y((I,i),c); \mathrm{El}(a)\overline{\gamma'})$$

By Lemma 4.29 and the way $\mathbb{I}^{\mathbb{C}}$ is defined we have a canonical isomorphism $y((I,i),c) \cong y(I,c) \times \mathbb{I}^{\mathbb{C}}$. We now apply $c'_{I,i,c,\gamma}$ to the path $\delta = \lambda(i : \mathbb{I}^{\mathbb{C}}).(\rho, i)$ to get the element

$$\begin{aligned} \rho : y(I,c) \quad &\vdash \quad c'_{I,i,c,\gamma}\delta \\ &: \quad \Pi(\varphi : \mathbb{F})(u : \Pi(i : \mathbb{I}). \\ &\qquad [\varphi] \to B(\delta(i))).B(\delta(0))[\varphi \mapsto u(0)] \to B(\delta(1))[\varphi \mapsto u(1)], \end{aligned}$$

where $B = \mathrm{El}(a)\overline{\gamma'}$.

From this element we can define $c_{I,c}$ by using the Yoneda lemma again to get the element $\overline{c'_{I,i,c,\gamma}}$ of type

$$\Pi(\varphi : \mathbb{F})(u : \Pi(i : \mathbb{I}).[\varphi] \to B(\delta(i))).B(\delta(0))[\varphi \mapsto u(0)] \to B(\delta(1))[\varphi \mapsto u(1)],$$

which is a type in context $y(I,c)$, at $(I,c), id_{I,c}$. To recap, the composition c will map $\gamma \in (\Gamma^{\mathbb{I}^{\mathbb{C}}})(I,c)$ to the element $\overline{c'_{I,i,c,\gamma}}$.

Lemma 4.32. *For any a and $\mathbf{c}$ of correct types we have*

$$\begin{aligned} \mathrm{Comp}(\langle\!|a, \mathbf{c}|\!\rangle) &= \mathbf{c} \\ \mathrm{El}(\langle\!|a, \mathbf{c}|\!\rangle) &= \mathrm{El}(a) \\ \langle\!|\mathrm{El}(a), \mathrm{Comp}(a)|\!\rangle &= a \end{aligned}$$

Assumption 4 is satisfied

Using Lemmas 4.25 and 4.28 we can define $\forall$ in $\widehat{\mathcal{C} \times \omega}$ as the inclusion of the $\forall$ from $\widehat{\mathcal{C}}$. Lemma 4.24 can then be used to show that the new $\forall$ is the right adjoint to the map $\varphi \mapsto \lambda\_.\varphi$.

4.C.5 A model of GCTT

The construction of the model of GCTT uses the internal language, in the form of *dependent predicate logic* with additional types, terms, and equalities corresponding to objects, arrows and properties of the particular category, of the presheaf topos $\widehat{\mathcal{C} \times \omega}$. Thus the internal language we use is an extension of the language $\mathcal{L}$ used above.

We define our model of GCTT as an extension of the model of CTT from section 4.C.2. Therefore we only need to show how to interpret the new rules of GCTT, i.e., the ones that have to do with *guarded recursive types*.

The functor $\triangleright$

We first define $\triangleright$ on $\widehat{\mathbb{C} \times \omega}$ and then extend it to types in context and delayed substitutions.

Given $X \in \widehat{\mathbb{C} \times \omega}$ we define

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

with restrictions inherited from X as in if $(f, n \leq m) : (I,n) \to (J,m)$ then

$$\triangleright X(f, n \leq m) : X(J,m) \to X(I,n)$$

$$\triangleright X(f, n \leq m) = \begin{cases} ! & \text{if } n = 1 \\ X(f, k \leq m-1) & \text{if } n = k+1 \end{cases}$$

where $n \leq m$ is the unique morphism $n \to m$ (and similarly $k \leq m-1$), and $!$ as the unique morphism into 1, the chosen singleton set.

There is a natural transformation

$$\begin{aligned} \text{next} &: id_{\widehat{\mathbb{C} \times \omega}} \to \triangleright \\ (\text{next}_X)_{I,0} &= ! \\ (\text{next}_X)_{I,n+1} &= X\,(id_I, (n \leq n+1)). \end{aligned}$$

Lemma 4.33. *The functor $\triangleright$ is continuous.*

Proof. Limits in presheaf toposes are computed pointwise. Limit of any diagram of terminal objects is the terminal object. □

Lemma 4.34. *For any X and any morphism $\alpha : \triangleright X \to X$ there exists a unique global element $\beta : 1 \to X$ such that*

$$\alpha \circ \text{next} \circ \beta = \beta.$$

Hence the triple $(\widehat{\mathbb{C} \times \omega}, \triangleright, \text{next})$ is a model of guarded recursive terms [14, Definition 6.1].

Proof. Any global element β satisfying the fixed-point equation must satisfy the following two equations

$$\begin{aligned} \beta_{I,0}(\star) &= \alpha_{I,0}(\star) \\ \beta_{I,n+1}(\star) &= \alpha_{I,n+1}\,(\beta_{I,n}(\star)). \end{aligned}$$

Hence define β recursively on n. It is then easy to see by induction on n that β is a global element and that it satisfies the fixed-point equation. □

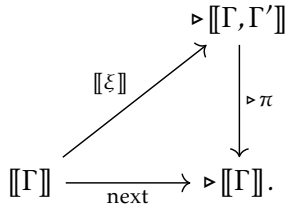
Using [14, Theorem 6.3] $\triangleright$ extends to all slices of $\widehat{\mathbb{C} \times \omega}$ and contractive morphisms on slices have unique fixed-points.

Remark 4.35. The construction in [14] ignores coherence issues, and there are no delayed substitutions, so we will define $\triangleright$ for types in contexts again, but the theorem cited gives us assurance that $\triangleright$ and fixed points exist in all slices. Moreover inspection of the construction in the cited paper shows that we are defining the correct notion, up to equivalent presentation of slices.

Delayed substitutions Semantically a *delayed* substitution

$$\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$$

will be interpreted as a morphism $[\![\xi]\!] : [\![\Gamma]\!] \to \triangleright[\![\Gamma, \Gamma']\!]$ making the following diagram commute



Here $\pi : [\![\Gamma, \Gamma']\!] \to [\![\Gamma]\!]$ is the composition of projections of the form

$$[\![\Gamma, \Gamma'', x : A]\!] \to [\![\Gamma, \Gamma'']\!].$$

In particular, if Γ' is the empty context then $\pi = id_{[\![\Gamma]\!]}$ and so $[\![\cdot]\!] = \text{next}$, where $\cdot$ is the empty delayed substitution.

Thus given a delayed substitution $\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$ and a type

$$\Gamma, \Gamma' \vdash A$$

define

$$\Gamma \vdash \triangleright\xi.A$$

to be

$$(\triangleright\xi.A)(I, n, \gamma) = \begin{cases} 1 & \text{if } n = 0 \\ A\left(I, m, [\![\xi]\!]_{I,n}(\gamma)\right) & \text{if } n = m + 1 \end{cases}$$

Note that this is exactly like substitution $A\xi$, except in the case where $n = 0$.

In turn, we interpret the rules

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

as follows. First, the empty delayed substitution is interpreted as next, as we already remarked above. Given $\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$ and $\Gamma \vdash t : \triangleright\xi.A$ define

$$[\![\vdash \xi\,[x \leftarrow t] : \Gamma \twoheadrightarrow \Gamma', x : A]\!]_{I,n}(\gamma) = \begin{cases} \star & \text{if } n = 0 \\ \left(\xi_{I,n}(\gamma), t_{I,n,\gamma}(\star)\right) & \text{otherwise} \end{cases}$$

The following two rules are easy to verify with the definitions we have provided.

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

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

The first rule follows immediately by observing that the interpretation of A in the extended context $\Gamma, \Gamma', x : B$ ignores the last component. The second rule follows because we exchange the same components in the interpretation of A as well as in the interpretation of the delayed substitution.

For the rest we need to define how next is interpreted.

Next

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

Given a term t and a delayed substitution ξ we define

$$[\![\mathsf{next}\,\xi.\,t]\!]_{I,n,\gamma}(\star) = \begin{cases} \star & \text{if } n = 0 \\ t_{I,m,[\![\xi]\!]_{I,n}(\gamma)}(\star) & \text{if } n = m + 1 \end{cases}$$

With this it is easy to see by computation that the rule

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

is sound.

$\triangleright$ and Π As we mentioned above a delayed substitution ξ is a morphism $\Gamma \to \triangleright(\Gamma, \Gamma')$. Hence we can treat it as a term of type $\triangleright(\Gamma, \Gamma')$ in context Γ. Further given a morphism $\gamma : \mathbb{I}^\omega \to \Gamma$ we can form the morphism

$$\xi \circ \gamma : \mathbb{I}^\omega \to \triangleright(\Gamma, \Gamma').$$

Finally by using Proposition 4.40 we can transport $\xi \circ \gamma : \mathbb{I}^\omega \to \triangleright(\Gamma, \Gamma')$ to a term

$$\overline{\xi \circ \gamma} : \triangleright(\mathbb{I}^\omega \to \Gamma, \Gamma')$$

in the empty context.

Lemma 4.36. *Given γ and ξ as above then for any type $\Gamma, \Gamma' \vdash A$ we have the equality of types*

$$i : \mathbb{I}^\omega \vdash \triangleright \left[\gamma' \leftarrow \overline{\gamma \circ \xi}\right].A(\gamma'(i)) = \triangleright \xi\, \gamma(i).A(\gamma(i)).$$

Here $\xi\, \gamma(i)$ is the delayed substitution $\vdash: \mathbb{I}^\omega \nrightarrow \Gamma, \Gamma'$ *obtained by substitution in terms of ξ.*

Proof. Proof by computation. Requires the unfolding of the definition of the isomorphism in Proposition 4.40. □

Dependent products and "constant" types To define composition for the $\triangleright$ type we will need type isomorphism commuting $\triangleright$ and dependent products in certain cases. We start with a definition.

Definition 4.37. A type $\Gamma \vdash A$ is *constant with respect to ω* if for all $I \in \mathbb{C}, n \in \omega, \gamma \in \Gamma(I, n)$ and for all $m \leq n$ the restriction

$$A(I, n, \gamma) \to A(I, m, \Gamma(id_I, m \leq n)(\gamma))$$

is the *identity function* (in particular, the two sets are equal).

Below we will use the shorter notation $\gamma_{\restriction_m}$ for $\Gamma(id_I, \leq n)(\gamma)$.

Note that this is a direct generalisation of "being constant" (being in the image of π^*) for presheaves (i.e., closed types). We have the following easy, but important, lemma.

Lemma 4.38. *Being constant with respect to ω is closed under substitution. If $\Gamma \vdash A$ is constant and $\rho : \Gamma' \to \Gamma$ is a context morphism then $\Gamma' \vdash A\rho$ is constant.*

Lemma 4.39. *Let X be a presheaf in the* essential image *of π^*. The identity type $x : X, y : X \vdash \mathrm{Id}_X(x, y)$ is* constant *with respect to ω.*

Proof. Recall that we have for $\gamma, \gamma' \in X(I, n)$.

$$(\mathrm{Id}_X(x, y))(I, n, \gamma, \gamma') = \begin{cases} 1 & \text{if } \gamma = \gamma' \\ \emptyset & \text{otherwise} \end{cases}$$

Thus for any $m \leq n$

$$(\mathrm{Id}_X(x, y))(I, m, \gamma_{\restriction_m}, \gamma'_{\restriction_m}) = \begin{cases} 1 & \text{if } \gamma_{\restriction_m} = \gamma'_{\restriction_m} \\ \emptyset & \text{otherwise} \end{cases}$$

But since $\cdot_{\restriction_m}$ is an isomorphism we have $\gamma_{\restriction_m} = \gamma'_{\restriction_m}$ if and only if $m = m'$, which concludes the proof. Since all the sets are singletons or empty the relevant restriction is then trivially the identity function. □

Using the assumptions stated above we have the following proposition.

Proposition 4.40. *Assume*

$$\Gamma \vdash A$$
$$\Gamma, \Gamma', x : A \vdash B$$
$$\vdash \xi : \Gamma \to\!\!\triangleright \Gamma'$$

and further that A is constant with respect to ω.

The canonical morphism from left to right in

$$\Gamma \vdash \triangleright\xi.\Pi(x : A).B \cong \Pi(x : A).\triangleright\xi.B \tag{4.6}$$

is an isomorphism. The canonical morphism is derived from the term

$$\lambda f.\lambda x.\,\mathsf{next}[\xi, f' \leftarrow f].\, f'x.$$

Proof. We need to establish an isomorphism of two presheaves on the category of elements of Γ. Since we already have one of the directions we will first define the other direction explicitly. We define $F : \Pi(x : A).\triangleright\xi.B \to \triangleright\xi.\Pi(x : A).B$. Let $I \in \mathbb{C}$, $n \in \omega$ and $\gamma \in \Gamma(I, n)$. Take $\alpha \in (\Pi(x : A).\triangleright\xi.B)(I, n, \gamma)$. If $n = 0$ then we have only one choice.

$$F_{I,0,\gamma}(\alpha) = \star$$

So assume that $n = m + 1$. Then we need to provide an element of

$$F_{I,n,\gamma}(\alpha) \in (\Pi(x : A).B)(I, m, \xi_{I,n}(\gamma)).$$

Which means that for each $f : J \to I$ and each $k \leq m$ we need to give a dependent function

$$\beta_{f,k} : (a \in A(J, k, (\Gamma, \Gamma')(f, k \leq m)(\xi_{I,n}(\gamma)))) \to B(J, k, (\Gamma, \Gamma')(f, k \leq m)(\xi_{I,n}(\gamma)), a)$$

Because $\Gamma \vdash A$ we have

$$A(J, k, (\Gamma, \Gamma')(f, k \leq m)(\xi_{I,n}(\gamma))) = A\left(J, k, \pi_{J,k}((\Gamma, \Gamma')(f, k \leq m)(\xi_{I,n}(\gamma)))\right)$$

where $\pi : \Gamma, \Gamma' \to \Gamma$ is the composition of projections. By naturality we have

$$\pi_{J,k}((\Gamma, \Gamma')(f, k \leq m)(\xi_{I,n}(\gamma))) = \Gamma(f, k \leq m)(\pi_{I,m}(\xi_{I,n}(\gamma))).$$

Now $\pi_{I,m} = \triangleright(\pi)_{I,n}$ and so we have (because ξ is a delayed substitution)

$$\pi_{I,m}(\xi_{I,n}(\gamma)) = \mathrm{next}(\gamma)_{I,n} = \Gamma(id_I, m \leq n)(\gamma).$$

Hence we have

$$A(J, k, (\Gamma, \Gamma')(f, k \leq m)(\xi_{I,n}(\gamma))) = A(J, k, \Gamma(f, k \leq n)(\gamma)).$$

And because A is *constant* we further have

$$A(J,k,\Gamma(f,k\le n)(\gamma)) = A(J,k+1,\Gamma(f,k+1\le n)(\gamma))$$

(by assumption $k \le m$ and $n = m+1$.

Now $\alpha_{f,k+1}$ is a dependent function

$$(a \in A(J,k+1,\Gamma(f,k+1\le n)(\gamma))) \to (\triangleright\xi.B)(J,k+1,\Gamma(f,k+1\le n)(\gamma),a)$$

And we have

$$\triangleright\xi.B)(J,k+1,\Gamma(f,k+1\le n)(\gamma),a) = B\Big(J,k,\xi_{J,k+1}(\Gamma(f,k+1\le n)(\gamma)),a\Big)$$

(because the relevant restriction of A is the identity). Now

$$\begin{aligned}\xi_{J,k+1}(\Gamma(f,k+1\le n)) &= (\triangleright(\Gamma,\Gamma'))(f,k+1\le n)(\xi_{I,n}(\gamma))\\ &= (\Gamma,\Gamma')(f,k\le m)(\xi_{I,n}(\gamma)).\end{aligned}$$

Thus, we can define

$$\beta_{f,k} = \alpha_{f,k+1}.$$

The fact that β is a natural family follows from the fact that α is a natural family. Naturality of F follows easily by the fact that restrictions for Π types are defined by "precomposition".

The fact that it is the inverse to the canonical morphism follows by a tedious computation. □

Corollary 4.41. *If $\Gamma \vdash \varphi : \mathbb{F}$ then we have an isomorphism of types*

$$\Gamma \vdash \triangleright\xi.\Pi(p:[\varphi]).B \cong \Pi(x:[\varphi]).\triangleright\xi.B. \tag{4.7}$$

Proof. Using Proposition 4.40 it suffices to show that $\Gamma \vdash [\varphi]$ is constant with respect to ω. Using Lemmas 4.38 and 4.39 it further suffices to show that the presheaf $\mathbb{F}$ is in the essential image of π^*, which is exactly what Lemma 4.28 states. □

Interpreting later types

Lemma 4.42. *Formation of $\triangleright\xi$-types preserves compositions. More precisely, 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 $\mathbf{c} : \Phi(\Gamma;\triangleright\xi.A)$.*

Note that the types in Γ' need not be fibrant.

Proof. We introduce the following variables:

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

Using Lemma 4.36 we can rewrite the types of u and a_0:

$$\begin{aligned}u &: \Pi(i : \mathbb{I}).\, \left(\triangleright\left[\gamma' \leftarrow \overline{\xi \circ \gamma}\right].A(\gamma'\, i)\right)^{\varphi}\\ a_0 &: \triangleright\left[\gamma' \leftarrow \overline{\xi \circ \gamma}\right].A(\gamma'\, 0).\end{aligned}$$

Furthermore, we have the following type isomorphisms:

$$\begin{aligned}\Pi(i : \mathbb{I}).\, \left(\triangleright\left[\gamma' \leftarrow \overline{\xi \circ \gamma}\right].A(\gamma'\, i)\right)^{\varphi} &\cong \Pi(i : \mathbb{I}).\triangleright\left[\gamma' \leftarrow \overline{\xi \circ \gamma}\right].(A(\gamma'\, i))^{\varphi} && \text{(Corr. 4.41)}\\ &\cong \triangleright\left[\gamma' \leftarrow \overline{\xi \circ \gamma}\right].\Pi(i : \mathbb{I}).\, (A(\gamma'\, i))^{\varphi}, && \text{(Prop. 4.40)}\end{aligned}$$

which means that we have a term

$$\tilde{u} : \triangleright\left[\gamma' \leftarrow \overline{\xi \circ \gamma}\right].\Pi(i : \mathbb{I}).\, (A(\gamma'\, i))^{\varphi}.$$

We can now – almost – form the term

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

In order for the composition sub-term to be well-typed, we need that $a'_0 = u\, 0$ under the assumption φ. This is equivalent to saying that the type

$$\triangleright\begin{bmatrix}\gamma' \leftarrow \overline{\xi \circ \gamma}\\ u' \leftarrow \tilde{u}\\ a'_0 \leftarrow a_0\end{bmatrix}.(\mathrm{Id}(a'_0, u'\, 0))^{\varphi}$$

is inhabited. We transform the type as follows:

$$\begin{aligned}&\triangleright\begin{bmatrix}\gamma' \leftarrow \overline{\xi \circ \gamma}\\ u' \leftarrow \tilde{u}\\ a'_0 \leftarrow a_0\end{bmatrix}.(\mathrm{Id}(a'_0, u'\, 0))^{\varphi}\\ \cong\;& \left(\triangleright\begin{bmatrix}\gamma' \leftarrow \overline{\xi \circ \gamma}\\ u' \leftarrow \tilde{u}\\ a'_0 \leftarrow a_0\end{bmatrix}.\mathrm{Id}(a'_0, u'\, 0)\right)^{\varphi} && \text{(Corr. 4.41)}\\ =\;& \left(\mathrm{Id}\left(\mathsf{next}\begin{bmatrix}\gamma' \leftarrow \overline{\xi \circ \gamma}\\ u' \leftarrow \tilde{u}\\ a'_0 \leftarrow a_0\end{bmatrix}.a'_0, \mathsf{next}\begin{bmatrix}\gamma' \leftarrow \overline{\xi \circ \gamma}\\ u' \leftarrow \tilde{u}\\ a'_0 \leftarrow a_0\end{bmatrix}.u'\, 0\right)\right)^{\varphi}\\ =\;& (\mathrm{Id}(a_0, u\, 0))^{\varphi},\end{aligned}$$

where the last equality uses that $\tilde{u}$ is defined using the inverse of

$$\lambda f\, \lambda x.\, \mathsf{next}\, \xi\, [f' \leftarrow f].\, f'\, x$$

(Prop. 4.40). By assumption it is the case that $(\mathrm{Id}(a_0, u\, 0))^{\varphi}$ is inhabited, and therefore $(*)$ is well-defined. This concludes the existence part proof, as

$$\triangleright \left[\gamma' \leftarrow \overline{\xi \circ \gamma}\right].A(\gamma'\, 1) = (\triangleright \xi.A)(\gamma\, 1),$$

by Lemma 4.36.

We now have to show that the $(*)$ is equal to $u\, 1$ under the assumption of φ. Assuming φ, we get by the properties of $\mathbf{c}_A$ that

$$\mathsf{next} \begin{bmatrix} \gamma' \leftarrow \overline{\xi \circ \gamma} \\ u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{bmatrix}.\mathbf{c}_A\, \gamma'\, \varphi\, u'\, a'_0 = \mathsf{next} \begin{bmatrix} \gamma' \leftarrow \overline{\xi \circ \gamma} \\ u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{bmatrix}.u'\, 1,$$

and by the definition of $\tilde{u}$ (Prop. 4.40) we have that

$$\mathsf{next} \begin{bmatrix} \gamma' \leftarrow \overline{\xi \circ \gamma} \\ u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{bmatrix}.u'\, 1 = u\, 1$$

as desired. □

4.C.6 Summary of the semantics of GCTT

The interpretation of the syntax of GCTT follows the usual pattern for interpreting dependent type theory, see, e.g., the handbook chapter [74]: we define a partial function on raw types and terms and then show that it is defined and independent of the derivation on all derivable judgements.

In all we define the interpretations of the following judgements with the help of the internal language of $\widehat{\mathcal{C} \times \omega}$.

- $[\![\Gamma \vdash]\!]$
- $[\![\Gamma \vdash A]\!]$
- $[\![\Gamma \vdash t : A]\!]$
- $[\![\Gamma \vdash A = B]\!]$
- $[\![\Gamma \vdash t = s : A]\!]$
- $[\![\vdash \xi : \Gamma \to \triangleright \Gamma']\!]$
- $[\![\rho : \Gamma \to \Gamma']\!]$

where the last one is a context morphism.

In particular soundness of the interpretation states that if

$$[\![\Gamma \vdash A = B]\!]$$

then the types $[\![\Gamma \vdash A]\!]$ and $[\![\Gamma \vdash B]\!]$ are interpreted as the same object. We have an analogous result for the judgement

$$[\![\Gamma \vdash t = s : A]\!].$$

and the interpretation of terms $[\![\Gamma \vdash t : A]\!]$ and $[\![\Gamma \vdash s : A]\!]$.

Chapter 5

Reduction Semantics

This chapter consists of preliminary work on a dependent type theory with guarded recursive types and a reduction relation on terms and types. It is part of an ongoing collaboration with Patrick Bahr and Rasmus Møgelberg, in which we aim to show decidability of type checking of a guarded dependent type theory via a strong normalisation proof.

The design of the calculus presented in this paper has roots in both Bahr's and Møgelberg's reduction semantics for a simple type theory with guarded recursion (unpublished), and in the Haskell implementation of a type checker for the guarded cubical type theory of Chapter 4. It is from the implementation work that we get the idea of replacing the delayed substitutions with resources.

In the following presentation we do not include identity types. The intention is to first make a strong-normalisation argument independently of the underlying type theory (e.g. cubical type theory, observational type theory), and then hopefully at a later stage extend it to include identity types.

5.1 Guarded Recursive Types with Resources

The delayed substitutions introduced in Chapter 3 have demonstrated good expressivity. However, it is difficult to design a reduction relation which gives rise to the same equational theory as described in Chapter 3 and Chapter 4. It is, for example, not obvious how to find a common reduct for the two terms

$$\lambda x : \triangleright A, y : \triangleright B.\, \mathsf{next}^{\kappa} \left[\begin{array}{c} x' \leftarrow x \\ y' \leftarrow y \end{array} \right] .t\, x\, y, \qquad \lambda x : \triangleright A, y : \triangleright B.\, \mathsf{next}^{\kappa} \left[\begin{array}{c} y' \leftarrow y \\ x' \leftarrow x \end{array} \right] .t\, x\, y,$$

which would be equal by the exchange rule.

To solve this problem we will look at an alternative to delayed substitutions. We will take a more resource-oriented approach, inspired by the resource interpretation of substructural logics – in particular our approach

seems to have a close connection with *affine logic*. Affine logic is a substructural logic where contraction is disallowed; in our setting this corresponds to disallowing terms of type $\triangleright\triangleright A \to \triangleright A$.

The rough idea is to see the 'later' type former as a form of *lollipop* connective, forming a resource aware function type $\kappa \multimap A$ from a type of clock resources κ. Instead of writing $\kappa \multimap A$ we will suggestively write $\overset{\kappa}{\triangleright} A$. For the sake of presentation we will in the beginning assume that we have exactly one such clock κ available, so it will be sufficient to talk of $\triangleright A$. Later on we will expand with clock quantification.

To illustrate the idea we will first describe the typing rules for $\triangleright A$ in a simply typed setting with only one clock, using dual contexts (later we will see why this is not quite enough to achieve the properties we want). Since $\triangleright A$ is a form of function type, it needs a 'lambda'. We will denote this with next, and it will bind a *resource token*, r. The intuition is that once we go under a $\triangleright$, we are shifting our viewpoint to 'tomorrow' – therefore we are provided with a token we can use to 'advance the clock' one day (if I know that "tomorrow is my birthday" then, from the viewpoint of tomorrow, I can say "today is my birthday"). The typing rule for next will have the form:

$$\frac{\Gamma \mid \Xi, r : \kappa \vdash t : A}{\Gamma \mid \Xi \vdash \mathsf{next}\, r.t : \triangleright A}$$

where Γ is the usual variable context, and Ξ is the resource context. We will have no way of combining the resource tokens, so we can express application as a unary rule:

$$\frac{\Gamma \mid \Xi \vdash t : \triangleright A}{\Gamma \mid \Xi, r : \kappa \vdash t^r : A}.$$

Notice that the resource token r disappears from the context once it is used. This reflects that we can only advance the clock once each time you shift your viewpoint (if I know that "the day after tomorrow is my birthday" then, from the viewpoint of tomorrow, I can say "tomorrow is my birthday", but not "today is my birthday"). The application rule for regular function types will allow duplication of the resource context:

$$\frac{\Gamma \mid \Xi \vdash t : A \to B \qquad \Gamma \mid \Xi \vdash u : A}{\Gamma \mid \Xi \vdash t\, u : B}$$

In this way we can can use a resource token as many times as we want in the 'breadth' of the term, but only once in 'depth': We get one resource token r every time we 'go under a later', which in turn can be used to 'strip off a later'. Once we have used the token, we cannot use it again in a subterm. As an example of the kind of resource duplication which is allowed, consider the term encoding delayed application $\circledast$:

$$\lambda x, y.\, \mathsf{next}\, r.\, x^r\, y^r \;:\; \triangleright(A \to B) \to \triangleright A \to \triangleright B.$$

An example of a term which *isn't* allowed is $\lambda x : \overset{\kappa\kappa}{\blacktriangleright\blacktriangleright} A.\, \mathsf{next}\, r : \kappa.t^{rr}$.

Just as for the regular function type, we have β and η-rules for $\overset{\kappa}{\blacktriangleright} A$:

$$\begin{array}{rcll} (\mathsf{next}\, r : \kappa.t)^{r'} & \to & t[r'/r] & (\beta_{\blacktriangleright}) \\ \mathsf{next}\, r : \kappa.t^{r} & \to & t & (\eta_{\blacktriangleright}) \end{array}$$

5.1.1 Translating Delayed Substitutions

We can translate terms using delayed substitutions into terms in the resource calculus. Let GTT_1 be a simple type theory with guarded recursive types and delayed substitutions on next's, and GTT_2 be a similar type theory with guarded recursive types with resources as described above. For now we will ignore the details about the unique fixed-points forming the guarded recursive types. Define a translation T from terms of GTT_1 to terms of GTT_2 by induction on the terms. We describe only the non-trivial case of next:

$$T(\mathsf{next}\,[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n].u) = \mathsf{next}\, r.T(u)[T(t_1)^r/x_1, \ldots, T(t_n)^r/x_n].$$

Recall from previous chapters the defining equations of delayed substitutions:

$$\mathsf{next}\,\xi\,[x \leftarrow t].u = \mathsf{next}\,\xi.u, \text{ if } x \notin \mathrm{FV}(u) \qquad (\textsc{Weakening})$$

$$\mathsf{next}\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v = \mathsf{next}\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v \qquad (\textsc{Exchange})$$

$$\mathsf{next}\,\xi\,[x \leftarrow \mathsf{next}\,\xi.t].u = \mathsf{next}\,\xi.u[t/x] \qquad (\textsc{Force})$$

$$\mathsf{next}\,\xi\,[x \leftarrow t].x = t \qquad (\textsc{Eta})$$

If we instead define equality of terms $t = u$ in GTT_1 as equality of their translations $T(t) = T(u)$, where the latter equality is induced by the rewrite rules of GTT_2, then we can verify that the above equalities still hold:

- Weakening: If x does not occur free in u, then

  $$T(\mathsf{next}\,\xi\,[x \leftarrow t].u) = T(\mathsf{next}\,\xi.u),$$

  because the substitution $[T(t)/x]$ will have no effect.

- Exchange: The following substitutions are the same: $\sigma\,[T(t)/x, T(u)/y]\,\tau$ and $\sigma\,[T(u)/y, T(t)/x]\,\tau$, so therefore

  $$T(\mathsf{next}\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v) = T(\mathsf{next}\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v).$$

- Force: Let $\xi = [y_1 \leftarrow t_1, \ldots, y_n \leftarrow t_n]$, and $\sigma = [T(t_1)^r/y_1, \ldots, T(t_n)^r/y_n]$. Then $(T(\mathsf{next}\,\xi.t))^r = (\mathsf{next}\, r.T(t)\sigma)^r \to_{\beta_{\blacktriangleright}} T(t)\sigma$, so

  $$\begin{array}{rcl} T(\mathsf{next}\,\xi\,[x \leftarrow \mathsf{next}\,\xi.t].u) & = & \mathsf{next}\, r.T(u)\sigma[(T(\mathsf{next}\,\xi.t))^r/x] \\ & =_{\beta_{\blacktriangleright}} & \mathsf{next}\, r.T(u)\sigma[T(t)\sigma/x] \\ & = & T(\mathsf{next}\,\xi.u[t/x]) \end{array}$$

where the last equality follows from a simple induction proof.

- Eta: This follows from the $\eta_\triangleright$-rule:

$$\begin{aligned} T(\mathsf{next}\,\xi\,[x \leftarrow t].x) &= \mathsf{next}\, r.T(t)^r \\ &=_{\eta_\triangleright} T(t). \end{aligned}$$

This suggests that we can use guarded recursive types with resources to provide a computational interpretation of delayed substitutions.

5.1.2 Stream Examples

We will show examples of stream programming to illustrate how to use this calculus in practice.

We assume that we have a constructor $\mathsf{cons} : \mathsf{Nat} \to \triangleright\mathsf{Str} \to \mathsf{Str}$, and destructors $\mathsf{head} : \mathsf{Str} \to \mathsf{Nat}$ and $\mathsf{tail} : \mathsf{Str} \to \triangleright\mathsf{Str}$. Now introduce the abbreviation $n ::_r s$ for $\mathsf{cons}\, n (\mathsf{next}\, r.s)$. We can now revisit some programming examples from earlier chapters. Consider the following definitions of `map` and the stream `nats`:

```
map : (Nat → Nat) → Str → Str
map f s = f (head s) ::_r map^r f (tail s)^r

nats : Str
nats = 0 ::_r map succ nats^r
```

Notice how these programs closely resemble their 'unguarded' cousins in a Haskell-like language. An example with multiple resource tokens is the following implementation of the stream of Fibonacci numbers:

```
zipWith : (Nat → Nat → Nat) → Str → Str → Str
zipWith f s_1 s_2 = f (head s_1 s_2) ::_r
                    zipWith^r f (tail s_1)^r (tail s_2)^r

fib : Str
fib = 0 ::_r1 1 ::_r2 zipWith (+) fib^r1 (tail fib^r1)^r2
```

5.2 Guarded Dependent Types with Resources

In this section we define in detail a guarded dependent type theory with resources.

Conceptually, the step to dependent types is simple: instead of having the simple affine function type $\kappa \multimap A$, we will have a dependent function type $(r : \kappa) \multimap A$, which we denote $\triangleright r : \kappa.A$.

The syntax of the terms and types will be:

$$\begin{array}{rcl} s,t,u,A,B & ::= & \Pi x : A.B \mid \Sigma x : A.B \mid \triangleright\, r : \kappa.A \mid \forall \kappa.A \mid 1 \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \mathcal{U}_\Delta \mid \mathsf{El}(A) \\ & \mid & \hat{\Pi} x : A.B \mid \hat{\Sigma} x : A.B \mid \hat{\triangleright} r : \kappa.A \mid \hat{\forall} \kappa.A \mid \hat{1} \mid \hat{\mathsf{Bool}} \mid \hat{\mathsf{Nat}} \\ & \mid & x \mid \lambda x.t \mid t\,u \mid \langle t,u\rangle \mid \pi_1 t \mid \pi_2 t \\ & \mid & \mathsf{next}\, r : \kappa.t \mid t^r \mid \Lambda\kappa.t \mid t[\kappa] \\ & \mid & \mathsf{dfix}^\kappa t \mid \mathsf{unfold}_r t \mid \mathsf{fold}_r t \\ & \mid & \langle\rangle \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\, s\, t\, u \mid 0 \mid \mathsf{suc}\, t \mid \mathsf{rec}\, t\, u\, v \end{array}$$

Here κ ranges over the set CV of clock variables, whereas r ranges over the set $\mathrm{RV} \cup \{*_\kappa \mid \kappa \in \mathrm{CV}\}$ of resource variables and resource constants – except for resource binders (terms of the form $\mathsf{next}\, r : \kappa.t$, $\triangleright r : \kappa.A$, and $\hat{\triangleright} r : \kappa.A$) where r ranges over the set RV of label variables only.

Our typing judgements will carry four contexts, and have the following shape: $\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} A$. Here Γ is the regular typing context, Δ is the clock context, Ξ is the resource context, and $\mathcal{I}$ is a resource index. The resource index keeps track of which resource tokens have at most been used, and is used to enforce the 'affineness' of $\triangleright$. A resource context is a sequence of typings of the form $r : \kappa$, where r is a resource and κ is a clock. A typing context Γ is a set of typings of the form $x :_{\mathcal{I}} A$ – where x is a variable, A is a type and $\mathcal{I}$ is a resource index – such that there is at most one typing for every variable. A clock context Δ is a set of clock variables. For composing contexts and resource indices we use commas for denoting the disjoint union. In particular that means, if we write $\mathcal{I}, r$, then $r \notin \mathcal{I}$ – and similarly for clock, typing and resource contexts.

The domain $\mathsf{dom}(\Xi)$ of a resource context Ξ is the set of its resources, i.e. $\mathsf{dom}(\Xi) = \{r \mid r : \kappa \in \Xi\}$. We write $\mathcal{I} \vdash^{\Xi}$ to indicate that $\mathcal{I}$ is a valid resource index in the resource context Ξ, which means that $\mathcal{I} \subseteq \mathsf{dom}(\Xi)$. We write $\Xi \vdash_\Delta$ to indicate that Ξ is a well-formed resource context in Δ, which means that whenever $r : \kappa \in \Xi$ then also $\kappa \in \Delta$.

We write $\overset{\kappa}{\triangleright} A$ and $\mathsf{next}^\kappa\, t$ as a shorthand for $\triangleright r : \kappa.A$ and $\mathsf{next}\, r : \kappa.t$, respectively, where r does not occur freely in A and t, respectively.

The reduction relation is defined in Figure 5.1, and the judgements are defined in Figures 5.2 and 5.3.

Many of the typing rules are standard, so we will not discuss them. One notable difference from the presentation of the rules in the introduction of this chapter is the resource indices $\mathcal{I}$ which are tied to every type variable. This is used in the variable rule:

$$\frac{(x :_{\mathcal{J}} A) \in \Gamma \qquad \mathcal{J} \subseteq \mathcal{I} \qquad \Gamma \vdash^{\Xi}_{\Delta} \qquad \mathcal{I} \vdash^{\Xi}}{\Gamma \vdash^{\Xi}_{\Delta} x :_{\mathcal{I}} A}$$

The side conditions enforce that we cannot apply any resource from $\mathcal{I}$ to a term which has x in it. This is to disallow terms such as:

$$\mathsf{next}\, r : \kappa.\mathsf{fix}^\kappa \lambda x.x^r$$

where $\mathsf{fix}^\kappa = \lambda f.f(\mathsf{dfix}^\kappa f)$, which otherwise seems to inhabit $\overset{\kappa}{\triangleright} A$ for any type A. If we reduce this term we get a term which violates the rule that you cannot apply the same resource token multiple times in series:

$$\mathsf{next}\, r : \kappa.\mathsf{fix}^\kappa \lambda x.x^r \to \mathsf{next}\, r : \kappa.(\mathsf{dfix}^\kappa \lambda x.x^r)^r.$$

The reason that $\mathsf{next}\, r : \kappa.\mathsf{fix}^\kappa \lambda x.x^r$ is ill typed is that the lambda rule needs the x in the context and x^r to have the same resource index, which is impossible in this case, because we necessarily must add r to the latter resource index.

In the following sections we discuss how to form guarded recursive types, and how to work with coinduction via the special resource token $*_\kappa$.

5.2.1 Forming Guarded Recursive Types

Like in the dependent type theories of Chapter 3 and Chapter 4 we do not have separate type formers for guarded recursive types, instead we form such types using the guarded fixed point combinator on the universe. In the extensional type theory of Chapter 3 we effectively have guarded equi-recursive types. In the type theory of Chapter 4 a guarded recursive type is only path equal to its unfolding, and therefore we must here rely on transports for folding and unfolding.

The aim of the present work is to provide reduction semantics to a guarded recursive type theory without identity types, in the hope that it can later be combined with identity types à la cubical types, or observational type theory. Thus we cannot rely on the properties of extensional equality, or on the compositions of cubical types, and therefore we explicitly add folds and unfolds.

The fixed points which we want to fold and unfold, are necessarily defined in terms of the dfix^κ combinator. Arguably, the simplest type construction using dfix^κ is the following: Let F be given such that $\Gamma \vdash^\Xi_\Delta F :_\mathcal{I} \overset{\kappa}{\triangleright} \mathcal{U}_{\Delta'} \to \mathcal{U}_{\Delta'}$ and assume that $r \in \Xi$. Then we can form the two types:

$$\Gamma \vdash^\Xi_\Delta \mathsf{El}(F(\mathsf{dfix}^\kappa F)) :_{\mathcal{I},r} \mathsf{type},$$
$$\Gamma \vdash^\Xi_\Delta \mathsf{El}((\mathsf{dfix}^\kappa F)^r) :_{\mathcal{I},r} \mathsf{type},$$

which ought to be isomorphic. Therefore we will have the constructions $\mathsf{unfold}_r t$ and $\mathsf{fold}_r t$ witnessing this isomorphism:

$$\frac{\Gamma \vdash^\Xi_\Delta t :_\mathcal{I} \mathsf{El}((\mathsf{dfix}^\kappa F)^r) \qquad \Xi \vdash r : \kappa \qquad r \in \mathcal{I} \cup \{*_\kappa\}}{\Gamma \vdash^\Xi_\Delta \mathsf{unfold}_r\, t :_\mathcal{I} \mathsf{El}(F(\mathsf{dfix}^\kappa F))}$$

$$\frac{\Gamma \vdash^\Xi_\Delta t :_\mathcal{I} \mathsf{El}(F(\mathsf{dfix}^\kappa F)) \qquad \Xi \vdash r : \kappa \qquad r \in \mathcal{I} \cup \{*_\kappa\}}{\Gamma \vdash^\Xi_\Delta \mathsf{fold}_r\, t :_\mathcal{I} \mathsf{El}((\mathsf{dfix}^\kappa F)^r)}$$

$$
\begin{array}{rlll}
(\lambda x.t)s & \to t[s/x] & & \\
(\Lambda\kappa.t)[\kappa'] & \to t[\kappa'/\kappa] & & \\
(\mathsf{next}\, r' : \kappa.t)^{r} & \to t\,[r/r'] & r \in \mathrm{RV} \cup \{*_\kappa\} & (\beta_\triangleright) \\
\mathsf{next}\, r : \kappa.t^{r} & \to t & \text{if } r : \kappa \notin \mathsf{fr}(t) & (\eta_\triangleright) \\
(\mathsf{dfix}^{\kappa}\, t)^{*_\kappa} & \to t\,(\mathsf{dfix}^{\kappa}\, t) & & \\
\mathsf{fold}_r \mathsf{unfold}_r t & \to t & & (\textsc{fold-unfold}) \\
\mathsf{fold}_{*_\kappa}\, t & \to t & & (\textsc{fold}) \\
\mathsf{unfold}_{*_\kappa}\, t & \to t & & (\textsc{unfold}) \\
\pi_i \langle t_1, t_2 \rangle & \to t_i & & \\
\mathsf{if}\,\mathsf{true}\, t_1\, t_2 & \to t_1 & & \\
\mathsf{if}\,\mathsf{false}\, t_1\, t_2 & \to t_2 & & \\
\mathsf{rec}\, 0\, t\, s & \to t & & \\
\mathsf{rec}\,(\mathsf{suc}\, t_1)\, t_2\, t_3 & \to t_3\, t_1\,(\mathsf{rec}\, t_1\, t_2\, t_3) & & \\
(\Lambda\kappa.t[\kappa]) & \to t & \text{if } \kappa \notin \mathsf{fc}(t) & (\textsc{clock-eta}) \\
\mathsf{El}\left(\hat{\Pi} x : s.\, t\right) & \to \Pi x : \mathsf{El}(s).\, \mathsf{El}(t) & & \\
\mathsf{El}\left(\hat{\Sigma} x : s.\, t\right) & \to \Sigma x : \mathsf{El}(s).\, \mathsf{El}(t) & & \\
\mathsf{El}\left(\hat{\mathsf{Nat}}\right) & \to \mathsf{Nat} & & \\
\mathsf{El}\left(\hat{1}\right) & \to 1 & & \\
\mathsf{El}\left(\hat{\mathsf{Bool}}\right) & \to \mathsf{Bool} & & \\
\mathsf{El}\left(\hat{\forall}\kappa.t\right) & \to \forall\kappa.\mathsf{El}(t) & & \\
\mathsf{El}\,(\hat{\triangleright} r : \kappa.t) & \to \triangleright r : \kappa.\mathsf{El}(t) & &
\end{array}
$$

Note that the side condition $r : \kappa \notin \mathsf{fr}(t)$ in $(\beta_\triangleright)$ and $(\eta_\triangleright)$ is always met for well-typed terms.

Figure 5.1: Reduction relation.

Contexts:

$$\frac{\Xi \vdash_\Delta}{\cdot \vdash^\Xi_\Delta} \qquad \frac{\Gamma \vdash^\Xi_\Delta \qquad \Gamma \vdash^\Xi_\Delta A :_\mathcal{I} \mathsf{type} \qquad \mathcal{I} \vdash^\Xi}{\Gamma, x :_\mathcal{I} A \vdash^\Xi_\Delta}$$

Universes:

$$\frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \mathcal{U}_{\Delta'} :_\mathcal{I} \mathsf{type}} \qquad \frac{\Gamma \vdash^\Xi_\Delta A :_\mathcal{I} \mathcal{U}_{\Delta'}}{\Gamma \vdash^\Xi_\Delta \mathsf{El}(A) :_\mathcal{I} \mathsf{type}}$$

Type formations:

$$\frac{\Gamma, x :_\mathcal{I} A \vdash^\Xi_\Delta B :_\mathcal{I} \mathsf{type}}{\Gamma \vdash^\Xi_\Delta \Pi x : A. B :_\mathcal{I} \mathsf{type}} \qquad \frac{\Gamma \vdash^\Xi_{\Delta,\kappa} A :_\mathcal{I} \mathsf{type} \qquad \Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \forall \kappa. A :_\mathcal{I} \mathsf{type}}$$

$$\frac{\Gamma \vdash^{\Xi, r:\kappa}_\Delta A :_{\mathcal{I},r} \mathsf{type} \qquad \kappa \in \Delta \qquad \Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \triangleright r : \kappa. A :_\mathcal{I} \mathsf{type}} \qquad \frac{\Gamma, x :_\mathcal{I} A \vdash^\Xi_\Delta B :_\mathcal{I} \mathsf{type}}{\Gamma \vdash^\Xi_\Delta \Sigma x : A. B :_\mathcal{I} \mathsf{type}}$$

$$\frac{\Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta 1 :_\mathcal{I} \mathsf{type}} \qquad \frac{\Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \mathsf{Bool} :_\mathcal{I} \mathsf{type}} \qquad \frac{\Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \mathsf{Nat} :_\mathcal{I} \mathsf{type}}$$

Codes:

$$\frac{\Gamma, x :_\mathcal{I} \mathsf{El}(A) \vdash^\Xi_\Delta B :_\mathcal{I} \mathcal{U}_{\Delta'}}{\Gamma \vdash^\Xi_\Delta \hat{\Pi} x : A. B :_\mathcal{I} \mathcal{U}_{\Delta'}} \qquad \frac{\Gamma \vdash^\Xi_{\Delta,\kappa} A :_\mathcal{I} \mathcal{U}_{\Delta',\kappa}}{\Gamma \vdash^\Xi_\Delta \hat{\forall} \kappa. A :_\mathcal{I} \mathcal{U}_{\Delta'}}$$

$$\frac{\Gamma \vdash^{\Xi, r:\kappa}_\Delta A :_{\mathcal{I},r} \mathcal{U}_{\Delta'} \qquad \kappa \in \Delta' \qquad \Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \hat{\triangleright} r : \kappa. A :_\mathcal{I} \mathcal{U}_{\Delta'}} \qquad \frac{\Gamma, x :_\mathcal{I} \mathsf{El}(A) \vdash^\Xi_\Delta B :_\mathcal{I} \mathcal{U}_{\Delta'}}{\Gamma \vdash^\Xi_\Delta \hat{\Sigma} x : A. B :_\mathcal{I} \mathcal{U}_{\Delta'}}$$

$$\frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \hat{1} :_\mathcal{I} \mathcal{U}_{\Delta'}} \qquad \frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \hat{\mathsf{Bool}} :_\mathcal{I} \mathcal{U}_{\Delta'}} \qquad \frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash^\Xi_\Delta}{\Gamma \vdash^\Xi_\Delta \hat{\mathsf{Nat}} :_\mathcal{I} \mathcal{U}_{\Delta'}}$$

Figure 5.2: Context and type formation rules.

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} A \qquad A \leftrightarrow^{*} B \qquad \Gamma \vdash^{\Xi}_{\Delta} B :_{\mathcal{I}} \mathsf{type}}{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} B}$$

$$\frac{(x :_{\mathcal{J}} A) \in \Gamma \qquad \mathcal{J} \subseteq \mathcal{I} \qquad \Gamma \vdash^{\Xi}_{\Delta} \qquad \mathcal{I} \vdash^{\Xi}}{\Gamma \vdash^{\Xi}_{\Delta} x :_{\mathcal{I}} A} \qquad \frac{\Gamma, x :_{\mathcal{I}} A \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} B}{\Gamma \vdash^{\Xi}_{\Delta} \lambda x.t :_{\mathcal{I}} \Pi x : A.B}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \Pi x : A.B \qquad \Gamma \vdash^{\Xi}_{\Delta} s :_{\mathcal{I}} A}{\Gamma \vdash^{\Xi}_{\Delta} t\, s :_{\mathcal{I}} B[s/x]} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta,\kappa} t :_{\mathcal{I}} A \qquad \Gamma \vdash^{\Xi}_{\Delta}}{\Gamma \vdash^{\Xi}_{\Delta} \Lambda\kappa.t :_{\mathcal{I}} \forall\kappa.A}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \forall\kappa.A \qquad \kappa' \in \Delta}{\Gamma \vdash^{\Xi}_{\Delta} t[\kappa'] :_{\mathcal{I}} A[\kappa'/\kappa]} \qquad \frac{\Gamma \vdash^{\Xi,r:\kappa}_{\Delta} t :_{\mathcal{I},r} A \qquad \kappa \in \Delta \qquad \Gamma \vdash^{\Xi}_{\Delta}}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{next}\, r : \kappa.t :_{\mathcal{I}} \triangleright r : \kappa.A}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \triangleright r' : \kappa.A \qquad r : \kappa \in \Xi}{\Gamma \vdash^{\Xi}_{\Delta} t^{r} :_{\mathcal{I},r} A\,[r/r']} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta,\kappa'} t :_{\mathcal{I}} \triangleright r : \kappa'.A \qquad \Gamma \vdash^{\Xi}_{\Delta} \qquad \kappa \in \Delta}{\Gamma \vdash^{\Xi}_{\Delta} (t[\kappa/\kappa'])^{*_\kappa} :_{\mathcal{I}} A\,[\kappa/\kappa', *_\kappa/r]}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta}}{\Gamma \vdash^{\Xi}_{\Delta} \langle\rangle :_{\mathcal{I}} 1} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} A \qquad \Gamma, x :_{\mathcal{I}} A \vdash^{\Xi}_{\Delta} s :_{\mathcal{I}} B}{\Gamma \vdash^{\Xi}_{\Delta} \langle t, s\rangle :_{\mathcal{I}} \Sigma x : A.B} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \Sigma x : A.B}{\Gamma \vdash^{\Xi}_{\Delta} \pi_1\, t :_{\mathcal{I}} A}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \Sigma x : A.B}{\Gamma \vdash^{\Xi}_{\Delta} \pi_2\, t :_{\mathcal{I}} B[\pi_1\, t/x]} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta}}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{true} :_{\mathcal{I}} \mathsf{Bool}} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta}}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{false} :_{\mathcal{I}} \mathsf{Bool}}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \mathsf{Bool} \qquad \Gamma \vdash^{\Xi}_{\Delta} u :_{\mathcal{I}} A \qquad \Gamma \vdash^{\Xi}_{\Delta} v :_{\mathcal{I}} A}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{if}\, t\, u\, v :_{\mathcal{I}} A} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta}}{\Gamma \vdash^{\Xi}_{\Delta} 0 :_{\mathcal{I}} \mathsf{Nat}}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \mathsf{Nat}}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{suc}\, t :_{\mathcal{I}} \mathsf{Nat}} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \mathsf{Nat} \qquad \Gamma \vdash^{\Xi}_{\Delta} u :_{\mathcal{I}} A \qquad \Gamma \vdash^{\Xi}_{\Delta} v :_{\mathcal{I}} \mathsf{Nat} \to A \to A}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{rec}\, t\, u\, v :_{\mathcal{I}} A}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \overset{\kappa}{\triangleright} A \to A}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{dfix}^{\kappa}\, t :_{\mathcal{I}} \triangleright^{\kappa} A} \qquad \frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \mathsf{El}((\mathsf{dfix}^{\kappa}\, F)^{r}) \qquad \Xi \vdash r : \kappa \qquad r \in \mathcal{I} \cup \{*_\kappa\}}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{unfold}_r\, t :_{\mathcal{I}} \mathsf{El}(F\,(\mathsf{dfix}^{\kappa}\, F))}$$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} \mathsf{El}(F\,(\mathsf{dfix}^{\kappa}\, F)) \qquad \Xi \vdash r : \kappa \qquad r \in \mathcal{I} \cup \{*_\kappa\}}{\Gamma \vdash^{\Xi}_{\Delta} \mathsf{fold}_r\, t :_{\mathcal{I}} \mathsf{El}((\mathsf{dfix}^{\kappa}\, F)^{r})}$$

Figure 5.3: Typing rules.

The special resource $*_\kappa$ will be discussed in the following section. In order to prevent diverging reductions we will not include a β-reduction

$$\mathsf{unfold}_r \mathsf{fold}_r t \not\to t,$$

but we will include η-reduction

$$\mathsf{fold}_r \mathsf{unfold}_r t \to t.$$

This does not affect canonicity for base types, because the free r in fold_r and unfold_r ensures that these constructions will only appear under a next.

5.2.2 Coinductive Types

Coinductive types are defined using clock quantification, as in Chapter 3. However, instead of using the $\mathsf{prev}\,\kappa.t$ construction from the same chapter to remove $\triangleright$'s where permitted, we will use special *resource constants* $*_\kappa$. The application rule for $*_\kappa$

$$\frac{\Gamma \vdash^{\Xi}_{\Delta,\kappa'} t :_{\mathcal{I}} \triangleright r : \kappa'.A \qquad \Gamma \vdash^{\Xi}_{\Delta} \qquad \kappa \in \Delta}{\Gamma \vdash^{\Xi}_{\Delta} (t[\kappa/\kappa'])^{*_\kappa} :_{\mathcal{I}} A\,[\kappa/\kappa', *_\kappa/r]}$$

is different from the standard resource application rule, because we need to make sure that the context of the term which we apply $*_\kappa$ to is constant with respect to κ.

The purpose of applying $*_\kappa$ to a term is to *advance* it. This means that fixed points will be unfolded once:

$$(\mathsf{dfix}^\kappa\, t)^{*_\kappa} \to t\,(\mathsf{dfix}^\kappa\, t),$$

and since the type of a term is advanced too, we can erase some folds and unfolds:

$$\mathsf{fold}_{*_\kappa}\, t \to t$$
$$\mathsf{unfold}_{*_\kappa}\, t \to t.$$

Note that we do not require the use of a separate 'adv' operation on types as in Chapter 3 – instead we have the substitution $[*_\kappa/r]$.

5.2.3 Examples

In the following examples we will not distinguish between types and codes, thus leaving out the El's. We will write $\mathsf{fix}^\kappa x.t$ as an abbreviation of

$$(\lambda x.t)(\mathsf{dfix}^\kappa\, \lambda x.t).$$

Guarded streams. First we will define the type of guarded streams of natural numbers:

$$\mathsf{Str}^\kappa \triangleq \mathsf{fix}^\kappa S.\, \mathsf{Nat} \times \triangleright\, r : \kappa.S^r.$$

As usual, the head function is easily defined:

$$\mathsf{head}^\kappa \triangleq \pi_1 : \mathsf{Str}^\kappa \to \mathsf{Nat}.$$

To define the tail function it is not sufficient to use the second projection, because our guarded recursive types are not equi-recursive:

$$\lambda s.\pi_2 s : \mathsf{Str}^\kappa \to \triangleright r : \kappa.(\mathsf{dfix}^\kappa \lambda S.\, \mathsf{Nat} \times \triangleright\, r : \kappa.S^r)^r.$$

We need to make use of unfold_r:

$$\mathsf{tail}^\kappa \triangleq \lambda s.\, \mathsf{next}\, r : \kappa.\mathsf{unfold}_r(\pi_2 s)^r : \mathsf{Str}^\kappa \to \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa.$$

Conversely we need to use fold_r for defining the stream constructor cons^κ:

$$\mathsf{cons}^\kappa \triangleq \lambda n, s.\, \langle n, \mathsf{next}\, r : \kappa.\mathsf{fold}_r s^r \rangle : \mathsf{Nat} \to \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa \to \mathsf{Str}^\kappa.$$

Let us see how these terms behave. Let n and s be variables.

$$\begin{aligned}
\mathsf{head}^\kappa(\mathsf{cons}^\kappa\, n\, s) &\to^* \pi_1 \langle n, \mathsf{next}\, r : \kappa.\mathsf{fold}_r s^r \rangle \\
&\to n \\
&\not\to
\end{aligned}$$

$$\begin{aligned}
\mathsf{tail}^\kappa(\mathsf{cons}^\kappa\, n\, s) &\to^* \mathsf{next}\, r : \kappa.\mathsf{unfold}_r(\pi_2(\mathsf{cons}^\kappa\, n\, s))^r \\
&\to^* \mathsf{next}\, r : \kappa.\mathsf{unfold}_r(\mathsf{next}\, r' : \kappa.\mathsf{fold}_{r'} s^{r'})^r \\
&\to^* \mathsf{next}\, r : \kappa.\mathsf{unfold}_r \mathsf{fold}_r s^r \\
&\not\to
\end{aligned}$$

Here $\not\to$ means that no more reductions are possible. We will write $n ::_r s$ as an abbreviation of $\mathsf{cons}^\kappa\, n\, (\mathsf{next}\, r : \kappa.s)$. It is then immediate that the normal form of $\mathsf{tail}^\kappa(n ::_r s)$ will be

$$\mathsf{next}\, r : \kappa.\mathsf{unfold}_r \mathsf{fold}_r s.$$

We can now define the familiar map function:

$$\begin{aligned}
&\mathsf{map}^\kappa : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Str}^\kappa \to \mathsf{Str}^\kappa \\
&\mathsf{map}^\kappa \triangleq \lambda f.\mathsf{fix}^\kappa m.\lambda s.\, f(\mathsf{head}^\kappa s) ::_r m^r\, (\mathsf{tail}^\kappa s)^r,
\end{aligned}$$

and use it to define the guarded stream of natural numbers:

$$\mathsf{nats}^\kappa \triangleq \mathsf{fix}^\kappa s.0 ::_r \mathsf{map}^\kappa\, \mathsf{suc}\, s^r \;:\; \mathsf{Str}^\kappa.$$

Coinductive streams. We will define the coinductive type of streams by quantifying over the free clock of Str^κ:

$$\mathsf{Str} \triangleq \forall\kappa.\mathsf{Str}^\kappa.$$

Whenever we have a clock κ_0 available (we can safely assume that this *clock constant* is always available) we can get the head of a stream:

$$\begin{aligned}&\mathsf{head} : \mathsf{Str} \to \mathsf{Nat}\\ &\mathsf{head} \triangleq \lambda s.\,\mathsf{head}^{\kappa_0}\, s[\kappa_0].\end{aligned}$$

By using the special $*$-resource, we can define the tail function for coinductive streams:

$$\begin{aligned}&\mathsf{tail} : \mathsf{Str} \to \mathsf{Str}\\ &\mathsf{tail} \triangleq \lambda s.\Lambda\kappa.(\mathsf{tail}^\kappa\, s[\kappa])^{*_\kappa}.\end{aligned}$$

Let s be some term of type Str^κ with a free resource token r. Then

$$\begin{aligned}\mathsf{tail}(\Lambda\kappa.n ::_r s) &\to^* \Lambda\kappa.(\mathsf{tail}^\kappa(n ::_r s))^{*_\kappa}\\ &\to^* \Lambda\kappa.(\mathsf{next}\, r : \kappa.\mathsf{unfold}_r\mathsf{fold}_r s)^{*_\kappa}\\ &\to^* \Lambda\kappa.s[*_\kappa/r].\end{aligned}$$

We will use this reduction later.

The following function returns the n'th element of a stream, and is an example of a function which cannot be defined on guarded streams:

$$\begin{aligned}&\mathsf{nth} : \mathsf{Nat} \to \mathsf{Str} \to \mathsf{Nat}\\ &\mathsf{nth} \triangleq \lambda n.\,\mathsf{rec}\, n\, \mathsf{head}\, (\lambda m, f, s.f\ (\mathsf{tail}\, s)).\end{aligned}$$

We can now also construct acausal stream functions such as the one that filters away all the odd elements of a stream:

$$\begin{aligned}&\mathsf{every2nd}^\kappa : \mathsf{Str} \to \mathsf{Str}^\kappa\\ &\mathsf{every2nd}^\kappa \triangleq \mathsf{fix}^\kappa e.\lambda s.\,\mathsf{head}^\kappa\, s[\kappa] ::_r e^r\, (\mathsf{tail}(\mathsf{tail}\, s))\\ \\ &\mathsf{every2nd} : \mathsf{Str} \to \mathsf{Str}\\ &\mathsf{every2nd} \triangleq \lambda s.\Lambda\kappa.\,\mathsf{every2nd}^\kappa\, s.\end{aligned}$$

It is now possible to calculate what the second even natural number is. Since we can define the coinductive stream of natural numbers as $\Lambda\kappa.\,\mathsf{nats}^\kappa$, the stream of even natural numbers will be $\mathsf{every2nd}(\Lambda\kappa.\,\mathsf{nats}^\kappa)$. The reduction

will go as follows:

$$
\begin{aligned}
&\mathsf{nth}\ 1\ (\mathsf{every2nd}(\Lambda\kappa.\,\mathsf{nats}^{\kappa}))\\
&\to^{*} \mathsf{head}(\mathsf{tail}(\mathsf{every2nd}(\Lambda\kappa.\,\mathsf{nats}^{\kappa})))\\
&\to^{*} \mathsf{head}(\mathsf{tail}(\mathsf{every2nd}(\Lambda\kappa.0 ::_r \mathsf{map}^{\kappa}\ \mathsf{suc}\ (\mathsf{nats}^{\overset{\kappa}{\blacktriangleright}})^r)))\\
&\to^{*} \mathsf{head}(\mathsf{tail}(0 ::_r (\mathsf{every2nd}^{\overset{\kappa}{\blacktriangleright}})^r(\mathsf{tail}(\Lambda\kappa.\,\mathsf{map}^{\kappa}\ \mathsf{suc}\ \mathsf{nats}^{\kappa}))))\\
&\to^{*} \mathsf{head}(\Lambda\kappa.\,\mathsf{every2nd}^{\kappa}(\mathsf{tail}(\Lambda\kappa.\,\mathsf{map}^{\kappa}\ \mathsf{suc}\ \mathsf{nats}^{\kappa})))\\
&\to^{*} \mathsf{head}(\Lambda\kappa.\,\mathsf{every2nd}^{\kappa}(\mathsf{tail}(\Lambda\kappa.\mathsf{suc}\,0 ::_r ((\mathsf{map}\ \mathsf{suc})^{\overset{\kappa}{\blacktriangleright}})^r\,(\mathsf{tail}^{\kappa}\ \mathsf{nats}^{\kappa})^r)))\\
&\to^{*} \mathsf{head}(\Lambda\kappa.\,\mathsf{every2nd}^{\kappa}(\Lambda\kappa.\,\mathsf{map}^{\kappa}\ \mathsf{suc}\ (\mathsf{tail}^{\kappa}\ \mathsf{nats}^{\kappa})^{*_{\kappa}}))\\
&\to^{*} \mathsf{head}(\Lambda\kappa.\,\mathsf{every2nd}^{\kappa}(\Lambda\kappa.\,\mathsf{map}^{\kappa}\ \mathsf{suc}\ (\mathsf{map}^{\kappa}\ \mathsf{suc}\ \mathsf{nats}^{\kappa})))\\
&\to^{*} \mathsf{head}(\Lambda\kappa.\,\mathsf{every2nd}^{\kappa}(\Lambda\kappa.\,\mathsf{map}^{\kappa}\ \mathsf{suc}\ (\mathsf{suc}\,0 ::_r ((\mathsf{map}\ \mathsf{suc})^{\overset{\kappa}{\blacktriangleright}})^r\,(\mathsf{tail}^{\kappa}\ \mathsf{nats}^{\kappa})^r)))\\
&\to^{*} \mathsf{head}(\Lambda\kappa.\,\mathsf{every2nd}^{\kappa}(\Lambda\kappa.\mathsf{suc}\,\mathsf{suc}\,0 ::_r (\cdots)))\\
&\to^{*} \mathsf{head}(\Lambda\kappa.\mathsf{suc}\,\mathsf{suc}\,0 ::_r (\cdots))\\
&\to^{*} \mathsf{suc}\,\mathsf{suc}\,0
\end{aligned}
$$

where $F^{\overset{\kappa}{\blacktriangleright}} = \mathsf{dfix}^{\kappa} f$, if $F^{\kappa} = \mathsf{fix}^{\kappa} x.f\,x$, and $(\cdots)$ is a term which is irrelevant for this particular reduction.

5.3 Future and Ongoing Work

We have seen examples providing evidence that guarded dependent type theory with resources is weakly normalising. We conjecture that it enjoys even stronger properties, namely:

- It is *strongly normalising*: t will have no infinite reduction sequence, provided that it is well-typed.
- It satisfies *canonicity* (for base types): if $\vdash_{\Delta} t : \mathsf{Nat}$, then $t \to^{*} \mathsf{suc}^{n}0$ for some $n \in \mathbb{N}$.

A proof of strong normalisation would lead to a proof of decidability of type checking. Proving these properties is currently ongoing work. We believe that it would be helpful to have a denotational model of the type theory. This model will necessarily be quite intricate, since it needs to allow clock synchronisation and provide an interpretation of resource tokens.

Additionally, we conjecture that the following syntactical properties hold:

- *Subject reduction*: if $\Gamma \vdash^{\Xi}_{\Delta} t :_{\mathcal{I}} A$ and $t \to u$, then $\Gamma \vdash^{\Xi}_{\Delta} u :_{\mathcal{I}} A$.
- *Confluence*: if $u_1\ {}^{*}\!\leftarrow t \to^{*} u_2$, then $u_1 \to^{*} t'\ {}^{*}\!\leftarrow u_2$ for some t'.

The proof of subject reduction seems to be a simple induction proof, and by showing that the diamond property of a parallel reduction relation holds, we believe that we can prove confluence.

There is one equality of the extensional type theory of Chapter 3 which is not verified by the type theory of this chapter: TmEq-∀-fresh (see page 79) which asserts that the terms $t[\kappa]$ and $t[\kappa']$ are equal, provided that t has type $\forall\kappa.A$ where κ does not occur free in A. Since our reduction relation is untyped there is no way that it can equate these terms. Unfortunately this rule seems to be necessary when working with dependent coinductive types, such as covectors, because it is used to get a type isomorphism $\forall\kappa.A \cong A$ when κ is not free in A.

If the above syntactic properties are confirmed, I believe that the time would be ripe to experiment with implementing guarded recursive types in Agda.

Bibliography

[1] Andreas Abel.
MiniAgda: Integrating sized and dependent types.
In *Partiality and Recursion in Interactive Theorem Provers (PAR)*, pages 14–28, 2010.
Cited on page 65.

[2] Andreas Abel and Brigitte Pientka.
Wellfounded recursion with copatterns: A unified approach to termination and productivity.
In *International Conference on Functional Programming (ICFP)*, pages 185–196, 2013.
Cited on pages 8 and 82.

[3] Andreas Abel and Andrea Vezzosi.
A formalized proof of strong normalization for guarded recursive types.
In *Programming Languages and Systems (APLAS)*, pages 140–158, 2014.
Cited on pages 28, 63, 100, and 110.

[4] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer.
Copatterns: Programming infinite structures by observations.
In *Principles of Programming Languages (POPL)*, pages 27–38, 2013.
Cited on page 8.

[5] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra.
Observational equality, now!
In *PLPV*, PLPV '07, pages 57–68, New York, NY, USA, 2007. ACM.
ISBN 978-1-59593-677-6.
doi: 10.1145/1292597.1292608.
URL http://doi.acm.org/10.1145/1292597.1292608.
Cited on page 116.

[6] Andrew W. Appel and David A. McAllester.
An indexed model of recursive types for foundational proof-carrying code.
ACM Trans. Program. Lang. Syst., 23(5):657–683, 2001.
Cited on page 1.

[7] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon.
A very modal model of a modern, major, general type system.
In *Principles of Programming Languages (POPL)*, pages 109–122, 2007.
Cited on pages 1, 4, 20, 65, 68, and 72.

[8] Robert Atkey and Conor McBride.
Productive coprogramming with guarded recursion.
In *International Conference on Functional Programming (ICFP)*, pages 197–208, 2013.
Cited on pages 2, 8, 14, 15, 21, 26, 34, 63, 64, 68, 76, 77, 82, 100, and 116.

[9] Gavin M. Bierman and Valeria CV de Paiva.
On an intuitionistic modal logic.
Studia Logica, 65(3):383–416, 2000.
Cited on pages 21, 26, 47, and 64.

[10] Lars Birkedal and Rasmus Ejlers Møgelberg.
Intensional type theory with guarded recursive types qua fixed points on universes.
In *Logic in Computer Science (LICS)*, pages 213–222, 2013.
Cited on pages 2, 15, 72, 82, and 105.

[11] Lars Birkedal, Jan Schwinghammer, and Kristian Støvring.
A metric model of lambda calculus with guarded recursion.
In *Fixed Points in Computer Science (FICS)*, pages 19–25, 2010.
Cited on pages 6, 40, and 64.

[12] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg.
The category-theoretic solution of recursive metric-space equations.
Theoretical Computer Science, 411(47):4102–4122, 2010.
Cited on page 49.

[13] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang.
Step-indexed Kripke models over recursive worlds.
In *Principles of Programming Languages (POPL)*, pages 119–132, 2011.
Cited on pages 1, 100, and 110.

[14] Lars Birkedal, Rasmus E. Møgelberg, Jan Schwinghammer, and Kristian Støvring.
First steps in synthetic guarded domain theory: step-indexing in the topos of trees.
Logical Methods in Computer Science (LMCS), 8(4), 2012.
Cited on pages 2, 6, 21, 35, 37, 44, 45, 49, 64, 68, 81, 82, 100, and 141.

[15] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi.
Guarded Cubical Type Theory: Path Equality for Guarded Recursion.
In *Computer Science Logic (CSL)*, 2016.
Cited on pages 16, 66, and 99.

[16] Lars Birkedal, Derek Dreyer, Ralf Jung, Robbert Krebbers, Filip Sieczkowski, Kasper Svendsen, David Swasey, and Aaron Turon.
Iris: A Basis for Concurrent Reasoning, 2016.
URL http://plv.mpi-sws.org/iris/.
Cited on page 2.

[17] Garrett Birkhoff et al.
Rings of sets.
Duke Mathematical Journal, 3(3):443–454, 1937.
Cited on page 134.

[18] Aleš Bizjak and Rasmus Ejlers Møgelberg.
A model of guarded recursion with clock synchronisation.
In *MFPS*, volume 319, pages 83–101. Elsevier, 2015.
Cited on pages 16 and 116.

[19] Aleš Bizjak and Rasmus E. Møgelberg.
A model of guarded recursion with clock synchronisation.
In *Mathematical Foundations of Programming Semantics (MFPS)*, pages 83–101, 2015.
Cited on pages 27, 63, 68, 69, 76, 81, and 82.

[20] Aleš Bizjak, Lars Birkedal, and Marino Miculan.
A model of countable nondeterminism in guarded type theory.
In *Rewriting and Typed Lambda Calculi (RTA-TLCA)*, pages 108–123, 2014.
Cited on page 45.

[21] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal.
Guarded dependent type theory with coinductive types.
In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 20–35, 2016.
Cited on pages 54, 64, and 65.

[22] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal.
Guarded dependent type theory with coinductive types.
In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 20–35, 2016.
Cited on pages 15, 67, 100, 104, 105, and 114.

[23] Edwin Brady.
Idris, a general-purpose dependently typed programming language: Design and implementation.
J. Funct. Programming, 23(5):552–593, 2013.
Cited on page 67.

[24] Adam Chlipala.
Certified Programming with Dependent Types.
MIT Press, 2013.
Cited on page 11.

[25] Ranald Clouston and Rajeev Goré.
Sequent calculus in the topos of trees.
In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 133–147, 2015.
Cited on pages 44 and 47.

[26] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal.
Programming and reasoning with guarded recursion for coinductive types.
arXiv:1501.02925, 2015.
Cited on page 22.

[27] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal.
Programming and reasoning with guarded recursion for coinductive types.
In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 407–421, 2015.
Cited on pages 14, 19, 22, 63, 77, 82, and 100.

[28] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal.
The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types.
Logical Methods of Computer Science (LMCS), 12(3), 2016.
Cited on pages 14 and 19.

[29] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg.
Cubical type theory: a constructive interpretation of the univalence axiom.
In *Proceedings of TYPES*, 2015.
To appear.
Cited on pages 12, 16, 66, 83, 100, 101, 102, 110, 112, 113, 118, 126, 130, and 134.

[30] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith.

Implementing Mathematics with the Nuprl Proof Development System.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
ISBN 0-13-451832-2.
Cited on page 67.

[31] Thierry Coquand.
Infinite objects in type theory.
In *Types for Proofs and Programs (TYPES)*, pages 62–78, 1993.
Cited on pages 20 and 68.

[32] Thierry Coquand.
Internal version of the uniform Kan filling condition.
Unpublished, 2015.
URL http://www.cse.chalmers.se/~coquand/shape.pdf.
Cited on page 116.

[33] William H Cornish and Peter R Fowler.
Coproducts of de morgan algebras.
Bulletin of the Australian Mathematical Society, 16(01):1–13, 1977.
Cited on page 134.

[34] Nils A. Danielsson and Thorsten Altenkirch.
Subtyping, declaratively: An exercise in mixed induction and coinduction.
In *Mathematics of Program Construction (MPC)*, pages 100–118, 2010.
Cited on page 65.

[35] Rowan Davies and Frank Pfenning.
A modal analysis of staged computation.
Journal of the ACM, 48(3):555–604, 2001.
Cited on page 64.

[36] Derek Dreyer, Amal Ahmed, and Lars Birkedal.
Logical step-indexed logical relations.
Logical Methods in Computer Science (LMCS), 7(2), 2011.
Cited on page 1.

[37] Peter Dybjer.
Internal type theory.
In *TYPES '95*, pages 120–134. Springer, 1996.
Cited on pages 111 and 125.

[38] Peter Dybjer.
Inductive families.
Formal Aspects of Computing, 6:440–465, 1997.
Cited on page 10.

[39] Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks.
Mix-automatic sequences.
Fields Workshop on Combinatorics on Words, contributed talk., 2013.
Cited on page 20.

[40] Jörg Endrullis, Dimitri Hendriks, and Martin Bodin.
Circular coinduction in Coq using bisimulation-up-to techniques.
In *Interactive Theorem Proving (ITP)*, pages 354–369. Springer, 2013.
Cited on page 32.

[41] Eduarde Giménez.
Codifying guarded definitions with recursive schemes.
In *TYPES*, pages 39–59, 1995.
Cited on pages 4 and 68.

[42] Martin Hofmann.
Extensional constructs in intensional type theory.
Springer, 1997.
Cited on page 100.

[43] Martin Hofmann and Thomas Streicher.
Lifting Grothendieck universes.
Unpublished, 1999.
URL http://www.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf.
Cited on page 113.

[44] Simon Huber.
Canonicity for cubical type theory.
Unpublished, 2016.
URL http://arxiv.org/abs/1607.04156.
Cited on page 12.

[45] John Hughes, Lars Pareto, and Amr Sabry.
Proving the correctness of reactive systems using sized types.
In *Principles of Programming Languages (POPL)*, pages 410–423, 1996.
Cited on pages 8, 65, and 82.

[46] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau.
Extending type theory with forcing.
In *Logic in Computer Science (LICS)*, pages 395–404, 2012.
Cited on page 65.

[47] B. Jacobs.
Categorical Logic and Type Theory.
Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

Cited on pages 69 and 74.

[48] Bart Jacobs and Jan Rutten.
A tutorial on (co)algebras and (co)induction.
Bulletin-European Association for Theoretical Computer Science, 62:222–259, 1997.
Cited on page 10.

[49] Alan Jeffrey.
LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs.
In *Programming Languages meets Program Verification (PLPV)*, pages 49–60, 2012.
Cited on page 65.

[50] Wolfgang Jeltsch.
Towards a common categorical semantics for linear-time temporal logic and functional reactive programming.
In *Mathematical Foundations of Programming Semantics (MFPS)*, pages 229–242, 2012.
Cited on page 65.

[51] Peter T. Johnstone.
Sketches of an Elephant: A Topos Theory Compendium.
Number 44 in Oxford Logic Guides. OUP, 2002.
Cited on pages 118 and 135.

[52] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer.
Higher-order ghost state.
Submitted., 2016.
Cited on page 65.

[53] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky.
The simplicial model of univalent foundations.
arXiv:1211.2851, 2012.
Cited on page 113.

[54] Neelakantan R. Krishnaswami.
Higher-order functional reactive programming without spacetime leaks.
In *International conference on Functional programming (ICFP)*, pages 221–232, 2013.
Cited on page 63.

[55] Neelakantan R. Krishnaswami and Nick Benton.
A semantic model for graphical user interfaces.
In *International conference on Functional programming (ICFP)*, pages 45–57, 2011.

Cited on page 63.

[56] Neelakantan R. Krishnaswami and Nick Benton.
Ultrametric semantics of reactive programs.
In *Logic in Computer Science (LICS)*, pages 257–266, 2011.
Cited on pages 8, 21, 63, 64, and 68.

[57] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann.
Higher-order functional reactive programming in bounded space.
In *Principles of Programming Languages (POPL)*, pages 45–58, 2012.
Cited on page 63.

[58] Tadeusz Litak.
Constructive modalities with provability smack.
Author's cut, v. 2.03., 2014.
Cited on page 44.

[59] Saunders Mac Lane.
Categories for the working mathematician, volume 5.
Springer Science & Business Media, 1978.
Cited on page 135.

[60] Saunders Mac Lane and Ieke Moerdijk.
Sheaves in Geometry and Logic.
Springer, 1992.
doi: 10.1007/978-1-4612-0927-0.
Cited on pages 113 and 136.

[61] Saunders Mac Lane and Ieke Moerdijk.
Sheaves in geometry and logic: A first introduction to topos theory.
Springer, 2012.
Cited on pages 44 and 58.

[62] Per Martin-Löf.
An intuitionistic theory of types: predicative part.
In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.
Cited on page 102.

[63] The Coq development team.
The Coq proof assistant reference manual.
LogiCal Project, 2004.
URL http://coq.inria.fr.
Version 8.0.
Cited on pages 1, 4, 20, 54, 67, and 100.

[64] Conor McBride and Ross Paterson.
Applicative programming with effects.
Journal of Functional Programming, 18(1):1–13, 2008.
Cited on pages 6, 23, 24, 47, 68, 71, 83, and 105.

[65] Stefan Milius, Lawrence S. Moss, and Daniel Schwencke.
Abstract GSOS rules and a modular treatment of recursive definitions.
Logical Methods in Computer Science, 9(3), 2013.
Cited on page 55.

[66] Rasmus E. Møgelberg.
A type theory for productive coprogramming via guarded recursion.
In *Computer Science Logic and Logic in Computer Science (CSL-LICS)*, 2014.
Cited on pages 27, 38, 63, 68, 69, 76, 77, 82, and 100.

[67] Rasmus E. Møgelberg and Patrick Bahr.
Reduction semantics for guarded recursion.
Unpublished, 2016.
Cited on page 47.

[68] Rasmus Ejlers Møgelberg and Marco Paviotti.
Denotational semantics of recursive types in synthetic guarded domain theory.
In *Logic in Computer Science (LICS)*, 2016.
Cited on pages 2 and 17.

[69] Hiroshi Nakano.
A modality for recursion.
In *Logic in Computer Science (LICS)*, pages 255–266, 2000.
Cited on pages 2, 4, 14, 20, 29, 63, 68, and 100.

[70] Ulf Norell.
Towards a practical programming language based on dependent type theory.
PhD thesis, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, 2007.
Cited on pages 4, 54, 67, and 100.

[71] Ian Orton and Andrew M. Pitts.
Axioms for modelling cubical type theory in a topos.
In *Computer Science Logic (CSL)*, 2016.
Cited on page 116.

[72] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal.
A model of PCF in guarded type theory.
In *Mathematical Foundations of Programming Semantics (MFPS)*, 2015.
Cited on pages 2, 17, and 69.

[73] 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.
Cited on pages 110 and 118.

[74] A. M. Pitts.
Categorical logic.
In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.
ISBN 0-19-853781-6.
URL http://www.oup.co.uk/isbn/0-19-853781-6.
Cited on page 148.

[75] François Pottier.
A typed store-passing translation for general references.
In *Principles of Programming Languages (POPL)*, pages 147–158, 2011.
Cited on page 63.

[76] Dag Prawitz.
Natural Deduction: A Proof-Theoretical Study.
Dover Publications, 1965.
Cited on pages 26 and 27.

[77] Reuben N. S. Rowe.
Semantic Types for Class-based Objects.
PhD thesis, Imperial College London, 2012.
Cited on page 63.

[78] Jan J. M. M. Rutten.
Behavioural differential equations: a coinductive calculus of streams, automata, and power series.
Theoretical Computer Science, 308(1):1–53, 2003.
Cited on pages 15, 22, 54, and 55.

[79] Paula G. Severi and Fer-Jan J. de Vries.
Pure type systems with corecursion on streams: from finite to infinitary normalisation.
In *International conference on Functional programming (ICFP)*, pages 141–152, 2012.
Cited on page 63.

[80] Filip Sieczkowski, Aleš Bizjak, and Lars Birkedal.

ModuRes: A Coq library for modular reasoning about concurrent higher-order imperative programming languages.
In *Interactive Theorem Proving (ITP)*, pages 375–390, 2015.
Cited on pages 2 and 65.

[81] Bas Spitters.
Cubical sets as a classifying topos.
TYPES, 2015.
Cited on pages 113 and 116.

[82] Kasper Svendsen and Lars Birkedal.
Impredicative concurrent abstract predicates.
In *Programming Languages and Systems (ESOP)*, pages 149–168, 2014.
Cited on pages 65 and 82.

[83] Sergei Tabachnikov.
Dragon curves revisited.
The Mathematical Intelligencer, 1(36):13–17, 2014.
Cited on page 20.

[84] The Univalent Foundations Program.
Homotopy Type Theory: Univalent Foundations of Mathematics.
http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.
Cited on pages 12, 74, and 101.

[85] Hongwei Xi, Chiyan Chen, and Gang Chen.
Guarded recursive datatype constructors.
In *Principles of Programming Languages (POPL)*, pages 224–235, 2003.
Cited on page 2.