THE GUARDED LAMBDA-CALCULUS
PROGRAMMING AND REASONING WITH GUARDED RECURSION FOR COINDUCTIVE TYPES

RANALD CLOUSTON, ALEŠ BIZJAK, HANS BUGGE GRATHWOHL, AND LARS BIRKEDAL

Department of Computer Science, Aarhus University, Denmark
e-mail address: ranald.clouston@cs.au.dk

Department of Computer Science, Aarhus University, Denmark
e-mail address: abizjak@cs.au.dk

Department of Computer Science, Aarhus University, Denmark
e-mail address: hbugge@cs.au.dk

Department of Computer Science, Aarhus University, Denmark
e-mail address: birkedal@cs.au.dk

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.

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* [17] 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:

1998 ACM Subject Classification: F.3.3, D.3.3, F.3.2, F.3.1.

Key words and phrases: guarded recursion, coinductive types, typed lambda-calculus, denotational semantics, program logic.

This is a revised and extended version of a FoSSaCS 2015 conference paper [13].

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

```
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* [47]), 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 [20]:

```
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 [33], will fail to detect the difference between these programs, and reject both.

A more flexible approach, first suggested by Nakano [38], is to guarantee productivity via *types*. A new modality, for which we follow Appel et al. [3] 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 $\blacktriangleright$ also. For example $\mathsf{interleave}\,\mathsf{paperfolds}'\,\mathsf{toggle}$ becomes ill-formed, as the $\mathsf{paperfolds}'$ self-reference has type $\blacktriangleright \mathsf{Str}^{\mathsf{g}}\,\mathbf{N}$, rather than $\mathsf{Str}^{\mathsf{g}}\,\mathbf{N}$ as required, but $\mathsf{interleave}\,\mathsf{toggle}\,\mathsf{paperfolds}$ will be well-formed.

Adding $\blacktriangleright$ 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* [29]: 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 $\blacktriangleright$ in a controlled way. This is provided by the *clock quantifiers* of Atkey and McBride [4], 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 [4] by regarding the 'everything now' operator as a unary type-former, written $\blacksquare$ 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 [5]. We will discuss the trade-offs involved in this alternative presentation in our discussion of related work in Section 5.1.

Overview of our contributions. In Section 1 we present the guarded λ-calculus, more briefly referred to as the $\mathsf{g}\lambda$-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 we define denotational semantics in the topos of trees [7] 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 3 we show how the internal logic of the topos of trees induces the program logic $L\mathsf{g}\lambda$ for reasoning about the denotations of $\mathsf{g}\lambda$-programs. Given the adequacy of our semantics, this logic permits proofs about the operational behaviour of terms. In Section 4 we demonstrate the expressiveness of the $\mathsf{g}\lambda$-calculus by showing the definability of solutions to Rutten's behavioural differential equations [43], and show that $L\mathsf{g}\lambda$ can be used to reason about them, as an alternative to standard bisimulation-based arguments. In Section 5 we conclude with a discussion of related and further work.

This paper is based on a previously published conference paper [13], 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 [14].
- 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 3.2; and by proving a property of box^+ in Section 3.3.
- We present new examples in Example 1.11 which show that converting a program to type-check in the $\mathsf{g}\lambda$-calculus is not always straightforward.
- We give a more intuitive introduction to the logic $L\mathsf{g}\lambda$ in Section 3, 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 3.2 that reveal how the explicit substitutions of the $\mathsf{g}\lambda$-calculus interact with real substitutions.
- We present (slightly improved) results regarding total and inhabited types in the $\mathsf{g}\lambda$-calculus in Section 3.2 which previously appeared only in an appendix to the technical report. Relatedly, we have generalised the proof in Example 3.11.1 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 4 which previously appeared only in an appendix to the technical report.
- We conduct a much expanded discussion of related and further work in Section 5.

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]

1. 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.

1.1. 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* [34], 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 1.4 to allow a cleaner presentation of the core system.

Definition 1.1. *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 $\mathsf{box}\, t$.

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.

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

Definition 1.2. The *reduction rules* on closed $\mathsf{g}\lambda$-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

$$\mathsf{prev}[\vec{x} \leftarrow \vec{t}].(\mathsf{next}\,t) \;\mapsto\; t[\vec{t}/\vec{x}] \tag{1.1}$$

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 (1.1) 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 [34].

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

Definition 1.3. *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 1.4. *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 $\mathsf{g}\lambda$-terms cannot be identified with the order of the call-by-name reductions of the corresponding λ-calculus term with the novel connectives erased.

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

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

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

Figure 1: Type formation for the gλ-calculus

Definition 1.5. *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.

1.2. **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 1.6. Open gλ-*types* are defined by the grammar

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

Type formation rules are defined inductively by the rules of Figure 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'.

Definition 1.7. The *typing judgments* are given in Figure 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.

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash t : \blacksquare A}{\Gamma \vdash \mathsf{unbox}\, t : A}$$

Figure 2: Typing rules for the gλ-calculus

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 [4] (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 recalls (but is more general than) the 'essentially modal' condition in the natural deduction calculus of Prawitz [41] for the modal logic Intuitionistic S4 (IS4). The term calculus for IS4 of Bierman and de Paiva [5], 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 [5]; see this reference for more in-depth discussion of the issue, and in particular how a failure to account for this issue causes problems for the calculus of Prawitz [41]. Similar side-conditions have also caused problems in the closely related area of calculi with clocks – see the identification by Bizjak and Møgelberg [12] of a problem with the type theory presented in earlier work by Møgelberg [36].

Lemma 1.8 (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.

1.3. **Examples.** We may now present example gλ-programs and their typings. We will first give causal programs without use of the constant modality ■, 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 1.9.

(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 1.4.

(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 [2] 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 [38] 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 3.9, once we have introduced the program logic $L\mathsf{g}\lambda$ for reasoning about $\mathsf{g}\lambda$-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)

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

is captured by the term

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

[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.

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 – $\mathsf{iterate}$, which takes a function $A \to A$ and a head A, and produces a stream by applying the function repeatedly, and $\mathsf{interleave}$, which interleaves two streams – in the obvious ways:

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

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 $\mathsf{iterate}$) the function, and (in the case of $\mathsf{interleave}$) the second stream, are not used until the next time step. We could alternatively use the definitions

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

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:

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

The recursion variable s in $\mathsf{paperfolds}$ has type $\blacktriangleright\mathsf{Str}^{\mathsf{g}}\,\mathbf{N}$, which means it cannot be given as the second argument to $\mathsf{interleave}'$ – only the more general $\mathsf{interleave}$ will do. However the erroneous definition of the regular paperfolding sequence that replaced $\mathsf{interleave}\,\mathsf{toggle}\;s$ with $\mathsf{interleave}'\,s\;\mathsf{toggle}$ 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 $A \times \blacktriangleright\text{-}$. This contrasts with the usual case of streams, which are merely the final coalgebra for the functor $A \times \text{-}$; the initial algebra for this

functor is trivial. To see the dual structure of guarded recursive types, consider the functions[3]

$$\begin{array}{rcll} \mathsf{initial} & \triangleq & \mathsf{fix}\,\lambda g.\lambda f.\lambda s.f\langle \mathsf{hd}^{\mathsf{g}}\, s, g \circledast \mathsf{next}\, f \circledast \mathsf{tl}^{\mathsf{g}}\, s\rangle & : ((A \times \blacktriangleright B) \to B) \to \mathsf{Str}^{\mathsf{g}} A \to B \\ \mathsf{final} & \triangleq & \mathsf{fix}\,\lambda g.\lambda f.\lambda x.(\pi_1(fx)) :: (g \circledast \mathsf{next}\, f \circledast \pi_2(fx)) & : (B \to A \times \blacktriangleright B) \to B \to \mathsf{Str}^{\mathsf{g}} A \end{array}$$

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 move on to examples involving the prev (previous) term-former and constant modality ■.

Example 1.10.

(1) The ■ type-former lifts guarded recursive streams to coinductive streams, as we will make precise in Example 2.4. 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:

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

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 ▶ 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 1.9 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

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

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

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 $\mathsf{g}\lambda$-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 Endullis et al. [21], where a bit more thought is required:

Example 1.11.

(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 $\mathsf{Str}^{\mathsf{g}}(\mathbf{1}+\mathbf{1}) \to \mathsf{Str}^{\mathsf{g}}(\mathbf{1}+\mathbf{1})$. But it does not have type $\blacktriangleright\mathsf{Str}^{\mathsf{g}}(\mathbf{1}+\mathbf{1}) \to \mathsf{Str}^{\mathsf{g}}(\mathbf{1}+\mathbf{1})$ 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 $\mathsf{Str}^{\mathsf{g}}(\mathbf{1}+\mathbf{1}) \to \mathsf{Str}^{\mathsf{g}}(\mathbf{1}+\mathbf{1})$ 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))
```

1.4. **Sums and the Constant Modality.** Atkey and McBride's calculus with clocks [4] 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 1.12 (ref. Definitions 1.1, 1.2, 1.4, 1.7)**.** 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 1.3. We extend the definition of evaluation contexts with

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

Finally, we add the new typing judgment

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

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. Denotational Semantics and Normalisation

This section gives denotational semantics for gλ-types and terms, as objects and arrows in the topos of trees [7], 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.

2.1. The topos of trees.

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

Definition 2.1. 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 $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$:

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

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 $x \in X_n$, $f_i(x \restriction i)$ 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 2.2.

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

$$Z \xleftarrow{id_Z} Z \xleftarrow{id_Z} Z_3 \xleftarrow{id_Z} \cdots$$

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 $\Delta\mathbb{N}$;

[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 4.3

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 $\mathsf{unbox} : \blacksquare \mathrel{\dot{\to}} id_{\mathcal{S}}$ 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 $*$:

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

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 $\mathsf{next} : id_{\mathcal{S}} \mathrel{\dot{\to}} \blacktriangleright$ has, for each component X, next_1 uniquely defined and $\mathsf{next}_{i+1} = r_i^X$:

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

2.2. **Denotational Semantics.** We may now see how the $\mathsf{g}\lambda$-calculus can be interpreted soundly in the topos of trees.

Definition 2.3. 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. [7, 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. 2.2).

Example 2.4.

(1) $[\![\mathsf{Str}^{\mathsf{g}}\,\mathbf{N}]\!]$ 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, $[\![\mathsf{Str}\,\mathbf{N}]\!] = \Delta(\mathbb{N}^\omega)$, 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 $[\![\mathsf{Str}^{\mathsf{g}}\,\mathbf{N}]\!]$.

More generally, any polynomial functor F on **Set** can be assigned a $\mathsf{g}\lambda$-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 [36, Theorem 2]. Therefore $\blacksquare$ is the modality that takes us from guarded recursive constructions to coinductive constructions.

(2) $[\![\mathsf{CoNat}^{\mathsf{g}}]\!]$ 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 2.5. *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 2.6. *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 2.7. 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 $a \mapsto [\![\Gamma, x : A \vdash t : B]\!]_j(\gamma\restriction_j, a)$;

- $[\![t_1 t_2]\!]_i(\gamma) = ([\![t_1]\!]_i(\gamma)_i) \circ [\![t_2]\!]_i(\gamma)$;

$[\![\mathsf{fold}\, t]\!]$ and $[\![\mathsf{unfold}\, t]\!]$ are defined via composition with the isomorphisms of Lemma 2.5. $[\![\mathsf{next}\, t]\!]$ and $[\![\mathsf{unbox}\, t]\!]$ are defined by composition with the natural transformations introduced in Definition 2.2. 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 2.6;
- $[\![\mathsf{box}[x_1 \leftarrow t_1, \ldots].t]\!]_i(\gamma)_j = [\![t]\!]_j([\![t_1]\!]_i(\gamma), \ldots)$, again using Lemma 2.6;
- Let $[\![t]\!]_j([\![t_1]\!]_i(\gamma), \ldots, [\![t_n]\!]_i(\gamma))$ (which is well-defined by Lemma 2.6) 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 $[\![\mathsf{box}^+[\vec{x} \leftarrow \vec{t}].t]\!]_i(\gamma) \triangleq [a, d]$.

Lemma 2.8. *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 $[\![t[\vec{t}/\vec{x}]]\!]_i(\gamma) = [\![t]\!]_i([\![t_1]\!]_i(\gamma), \ldots, [\![t_m]\!]_i(\gamma))$.*

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

$\mathsf{next}\, t$: case $i = 1$ is trivial. $[\![\mathsf{next}\, t[\vec{t}/\vec{x}]]\!]_{i+1}(\gamma) = r_i^{[\![A]\!]} \circ [\![t[\vec{t}/\vec{x}]]\!]_{i+1}(\gamma)$ by definition, which is $r_i^{[\![A]\!]} \circ [\![t]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots)$ by induction, which is $[\![\mathsf{next}\, t]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots)$.

$[\![(\mathsf{prev}[\vec{y} \leftarrow \vec{u}].t)[\vec{t}/\vec{x}]]\!]_i(\gamma) = [\![\mathsf{prev}[\vec{y} \leftarrow \vec{u}[\vec{t}/\vec{x}]].t]\!]_i(\gamma)$, which by definition is equal to $[\![t]\!]_{i+1}([\![u_1[\vec{t}/\vec{x}]]\!]_i(\gamma), \ldots)$, which is $[\![t]\!]_{i+1}([\![u_1]\!]_i([\![t_1]\!]_i(\gamma), \ldots), \ldots)$ by induction, which is $[\![\mathsf{prev}[\vec{y} \leftarrow \vec{u}].t]\!]_i([\![t_1]\!]_i(\gamma), \ldots)$.

$u_1 \circledast u_2$: case $i = 1$ is trivial. $[\![(u_1 \circledast u_2)[\vec{t}/\vec{x}]]\!]_{i+1}(\gamma) = ([\![u_1[\vec{t}/\vec{x}]]\!]_{i+1}(\gamma)_i) \circ [\![u_2[\vec{t}/\vec{x}]]\!]_{i+1}(\gamma)$, which is $([\![u_1]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots)_i) \circ [\![u_2]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots)$, which is in turn equal to $[\![u_1 \circledast u_2]\!]_{i+1}([\![t_1]\!]_{i+1}(\gamma), \ldots)$.

$[\![\mathsf{box}[\vec{y} \leftarrow \vec{u}[\vec{t}/\vec{x}]].t]\!]_i(\gamma)_j = [\![t]\!]_j([\![u_1[\vec{t}/\vec{x}]]\!]_i(\gamma), \ldots)$, which is $[\![t]\!]_j([\![u_1]\!]_i([\![t_1]\!]_i(\gamma), \ldots), \ldots)$ by induction, which is $[\![\mathsf{box}[\vec{y} \leftarrow \vec{u}].t]\!]_i([\![t_1]\!]_i(\gamma), \ldots)_j$.

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

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

Theorem 2.9 (Soundness)**.** *If $t \rightsquigarrow u$ then $[\![t]\!] = [\![u]\!]$.*

Proof. We verify the reduction rules of Definition 1.2; extending this to any evaluation context, and to $\rightsquigarrow$, is easy. The product reduction case is standard, and function case requires Lemma 2.8. $\mathsf{unfold}\, \mathsf{fold}$ is the application of mutually inverse arrows.

$[\![\mathsf{prev}[\vec{x} \leftarrow \vec{t}].t]\!]_i = [\![t]\!]_{i+1}([\![t_1]\!]_i, \ldots)$. 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 $[\![t_k]\!]_i = [\![t_k]\!]_{i+1}$. $[\![t]\!]_{i+1}([\![t_1]\!]_{i+1}, \ldots) = [\![t[\vec{t}/\vec{x}]]\!]_{i+1}$ by Lemma 2.8, which is $[\![\mathsf{prev}\, t[\vec{t}/\vec{x}]]\!]_i$.

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

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 $([\![t_1]\!]_i)_i \circ [\![t_2]\!]_i = [\![t_1 t_2]\!]_i = [\![t_1 t_2]\!]_i \circ r_i^1 = r_i^{[\![B]\!]} \circ [\![t_1 t_2]\!]_{i+1} = [\![\mathsf{next}(t_1 t_2)]\!]_{i+1}$.

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

box^+-reduction: Because each $[\![A_k]\!]$ is a constant object (Lemma 2.6), $[\![t_k]\!]_i = [\![t_k]\!]_j$ for all i, j. Hence $[\![\mathsf{box}^+[\vec{x} \leftarrow \vec{t}].t]\!]_i$ is defined via components $[\![t]\!]_j([\![t_1]\!]_j, \ldots)$ and $[\![\mathsf{box}^+\, t[\vec{t}/\vec{x}]]\!]$ is defined via components $[\![t[\vec{t}/\vec{x}]]\!]_j$. These are equal by Lemma 2.8. $[\![\mathsf{box}^+\, \mathsf{in}_d\, t]\!]_i$ 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. **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. [8], 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 2.10. 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 2.11.

(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 2.12. 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 $aR_i^{A_d}u$.
- $fR_i^{A \to B}t$ 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 2.5);
- $aR_i^{\blacktriangleright A}t$ 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 2.11.2 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 2.11.1 for μ-types.

Lemma 2.13. *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 2.14. *$aR_{i+1}^A t$ 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 2.15. *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 2.16 (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 $\Gamma \vdash t : A$. $\langle\rangle$, 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 2.13 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 $aR_i^{A_d} u$. 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 2.13 completes.

$\lambda x.t$: Taking $j \leq i$ and $aR_j^A u$, 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 $[\![t]\!]_j(\vec{a}\restriction_j, a)$. For each k, $a_k \restriction_j R_j^{A_k} t_k$ by Lemma 2.14, 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 2.13 completes.

$\mathsf{unfold}\, t$: we reduce under unfold, then reduce $\mathsf{unfold}\, \mathsf{fold}$, then use Lemma 2.13.

$\mathsf{next}\, t$: Trivial for index 1. For $i = j + 1$, if each $a_k R_{j+1}^{A_k} t_k$ then by Lemma 2.14 $r_j^{[\![A_k]\!]}(a_k)R_j^{A_k} t_k$. 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 2.15. 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 2.13 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 2.15 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 2.13 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 2.15. 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 2.17 (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 2.16 to closed types. (2) and (3) hold by (1) and inspection of Definition 2.12. □

Definition 2.18. 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 2.19. *$[\![t]\!] = [\![u]\!]$ implies $t \simeq_{\mathsf{ctx}} u$.*

Proof. $[\![C[t]]\!] = [\![C[u]]\!]$ by compositionality of the denotational semantics. Then by Theorem 2.17.2 they reduce to the same value. □

3. 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 $L\mathsf{g}\lambda$ which supports reasoning about the contextual equivalence of $\mathsf{g}\lambda$-programs, remark on some properties of this program logic, and give some example proofs.

3.1. **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 [32]. The internal logic of $\mathcal{S}$ has been explored elsewhere [7, 15, 31], but to motivate the results of this section we make some observations here.

As discussed in Example 2.4.2, the subobject classifier Ω is exactly the denotation of the *guarded conatural numbers* $\mathsf{CoNat}^{\mathsf{g}}$, as defined in the g$\lambda$-calculus in Section 1.4. The propositional connectives can then be defined via gλ-functions on the guarded conaturals: false $\bot$ is cozero, as defined in Section 1.4; true $\top$ is $\mathsf{infinity}$; conjunction $\wedge$ is a minimum function readily definable on pairs of guarded conaturals; $\neg$ is

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

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 [32, 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 1.4. 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 2.2.3. 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

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

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

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

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' [10]. 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 3.1.

- 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 $y \restriction_m = x$, 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

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

[5] called succ by Birkedal et al. [7]; we avoid this because of the clash with the name for a term-former.

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.

Example 3.2. 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 $\Box\,\phi$ 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

$$\Box(n) \;=\; \begin{cases} \infty & n = \infty \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

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 $\mathsf{g}\lambda$-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 (3.1).

Note further than $\Box$ does not commute with substitution; in particular, given a substitution σ, $\Box(\phi\sigma)$ does not necessarily imply $(\Box\,\phi)\sigma$. 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 $L\mathsf{g}\lambda$:

Definition 3.3. $L\mathsf{g}\lambda$ 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 $\mathsf{g}\lambda$-calculus, interpreted in $\mathcal{S}$ as in Section 2.2, and further extended by the modalities $\triangleright, \Box$.

We write $\Gamma \mid \Xi \vdash \phi$ 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 3.4. *Let t_1 and t_2 be two $\mathsf{g}\lambda$ 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 2.19) then implies that t_1 and t_2 are contextually equivalent. □

3.2. **Properties of the Logic.** The definition of the logic $L\mathsf{g}\lambda$ 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 [15] 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 $L\mathsf{g}\lambda$ is considerably more expressive than this; for example it is not decidable [37]. In this section we will establish

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

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

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

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

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

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

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

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

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

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

Figure 3: Equations between gλ-terms in Lgλ. Types in $\vec{A}, \vec{B}, C, D$ are assumed constant. comp is the composition $\lambda x.\lambda y.\lambda z.x(yz) : (B \to C) \to (A \to B) \to (A \to C)$.

some reasoning principles for Lgλ, which will assist us in the next section in constructing proofs about gλ-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 gλ-constructs, sound in the model $\mathcal{S}$, as listed in Figure 3.

Many of the rules of Figure 3 are unsurprising, adding η-rules to the β-rules of Definition 1.2, noting only that in the case of $\blacktriangleright$ we use the rule of equation (1.1), because

we are here allowing the consideration of open terms. The reduction rule for $\circledast$ is joined by the 'composition' equality for applicative functors [34]. In addition to the β-rule for box^+ of Definition 1.12, 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 [5]

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

comes as a corollary.

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

Definition 3.5. An $\mathcal{S}$-object is *total and inhabited* if all its restriction functions are surjective, and all its sets are non-empty.

A $\mathsf{g}\lambda$-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:

Lemma 3.6. *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 $\mathsf{g}\lambda$-types are total and inhabited, as the next lemma and its corollary show:

Lemma 3.7. *Let $F : (\mathcal{S}^{op} \times \mathcal{S})^{n+1} \to \mathcal{S}$ be a* locally contractive [7, 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 $\mathcal{M}$ [7, Section 5]. $\mathcal{M}$ is known to be an M-category in the sense of Birkedal et al. [9] 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 $ti\mathcal{S}$. □

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

$$\frac{}{\Gamma, x : X \mid \triangleright(\forall y : Y, \phi(x,y)) \vdash \forall y : Y, \triangleright\phi(x,y)}\ \forall\triangleright \qquad \frac{}{\Gamma \mid \Xi, \phi \vdash \triangleright\phi} \qquad \frac{\star \in \{\wedge, \vee, \Rightarrow\}}{\Gamma \mid \triangleright(\phi \star \psi) \dashv\vdash \triangleright\phi \star \triangleright\psi}$$

$$\frac{\Gamma \mid \neg\neg\phi \vdash \psi}{\Gamma \mid \phi \vdash \Box\psi} \qquad \frac{\Gamma \mid \phi \vdash \Box\psi}{\Gamma \mid \neg\neg\phi \vdash \psi} \qquad \frac{\Gamma \mid \phi \vdash \psi}{\Gamma \mid \Box\phi \vdash \Box\psi} \qquad \frac{}{\Gamma \mid \Box\phi \vdash \phi} \qquad \frac{}{\Gamma \mid \Box\phi \vdash \Box\Box\phi}$$

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

Figure 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.

Corollary 3.8. *All* gλ*-types that do not have the empty type* $\mathbf{0}$ *in their syntax tree are total and inhabited.*

Proof. The μ-case is covered by Lemma 3.7, 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 $L\mathsf{g}\lambda$, some making use of the concept of total and inhabited type, are listed in Figure 4, and in the lemmas below, whose proofs are all routine. Note that the rule $\text{EQ}^{\triangleright}_{\mathsf{next}}$ establishes a close link between $\blacktriangleright$ and $\triangleright$, as Lemma 3.10 does for $\blacksquare$ and $\Box$.

Lemma 3.9. *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 $L\mathsf{g}\lambda$ we can prove the proposition below.

Lemma 3.10. *The formula* $\Box(\mathsf{unbox}\, x =_A \mathsf{unbox}\, y) \Rightarrow x =_{\blacksquare A} y$ *is valid.* □

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

Example 3.11.

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

$$(\mathsf{map}^{\mathsf{g}}\, f) \circ (\mathsf{map}^{\mathsf{g}}\, g) =_{\mathsf{Str}^{\mathsf{g}} A \to \mathsf{Str}^{\mathsf{g}} C} \mathsf{map}^{\mathsf{g}}(f \circ g). \tag{3.2}$$

Equality of functions is extensional, so it suffices to show that these are equal on any stream of type $\mathsf{Str}^{\mathsf{g}} A$, 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 (3.2) with $\triangleright$ in front:

$$\triangleright((\mathsf{map}^{\mathsf{g}}\, f) \circ (\mathsf{map}^{\mathsf{g}}\, g) = \mathsf{map}^{\mathsf{g}}(f \circ g)). \tag{3.3}$$

Now unfolding the left hand side of (3.2) applied to s, using the definition of $\mathsf{map}^{\mathsf{g}}$ from Example 1.9.5, along with β-rules and Lemma 3.9, we get

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

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

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

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

$$f(g(\mathsf{hd}^{\mathsf{g}}\, s)) :: (\mathsf{next}((\mathsf{map}^{\mathsf{g}}\, f) \circ (\mathsf{map}^{\mathsf{g}}\, g)) \circledast \mathsf{tl}^{\mathsf{g}}\, s) \tag{3.4}$$

Unfolding the right of (3.2) similarly, we get

$$f(g(\mathsf{hd}^{\mathsf{g}}\, s)) :: (\mathsf{next}(\mathsf{map}^{\mathsf{g}}(f \circ g)) \circledast \mathsf{tl}^{\mathsf{g}}\, s) \tag{3.5}$$

These streams have the same head; we proceed on the tail using our induction hypothesis (3.3). 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 (3.4) equal to (3.5); 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

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

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

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

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

$$\triangleright(\forall s : \mathsf{Str}\, \mathbf{N}, P_{\mathsf{Str}^{\mathsf{g}}}(s) \Rightarrow Q_{\mathsf{Str}^{\mathsf{g}}}(\mathsf{map}^{\mathsf{g}}\, f\, s)) \tag{3.6}$$

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 3.8. 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 (3.6) and Löb finish the proof.

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

Example 3.12.

(1) Recall the functions $\mathsf{iterate}' : (A \to A) \to A \to \mathsf{Str}^{\mathsf{g}} A$ of Example 1.9.6 and $\mathsf{every2nd} : \mathsf{Str} A \to \mathsf{Str}^{\mathsf{g}} A$ of Example 1.10.3. 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 $\lambda x.f(fx)$.

First we prove the intermediate result

$$\mathsf{tl}(\mathsf{box}\, \iota.\, \mathsf{iterate}'\; f\, x) =_{\mathsf{Str} A} \mathsf{box}\, \iota.\, \mathsf{iterate}'\; f\, (f\, x) \tag{3.7}$$

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 3. The remaining steps follow from unfolding definitions, various β-rules, and Lemma 3.9.

Now for Löb induction assume

$$\triangleright\left(\mathsf{every2nd}(\mathsf{box}\,\iota.\,\mathsf{iterate}'\ f\,x) =_{\mathsf{Str}^{\mathsf{g}}A}\ \mathsf{iterate}'\ f^2\,x\right), \tag{3.8}$$

then we can derive

$$\begin{aligned}
&\mathsf{every2nd}\,(\mathsf{box}\,\iota.\,\mathsf{iterate}'\ f\,x) \\
&\qquad = x :: (\mathsf{next}\,\mathsf{every2nd}) \circledast (\mathsf{next}\,\mathsf{tl}\,\mathsf{tl}\,\mathsf{box}\,\iota.\,\mathsf{iterate}'\ f\,x) \\
&\qquad = x :: \mathsf{next}\,\mathsf{every2nd}\,\mathsf{tl}\,\mathsf{tl}\,\mathsf{box}\,\iota.\,\mathsf{iterate}'\ f\,x \\
&\qquad = x :: \mathsf{next}\,\mathsf{every2nd}\,\mathsf{box}\,\iota.\,\mathsf{iterate}'\ f(f^2\,x) && (3.7) \\
&\qquad = x :: \mathsf{next}\,\mathsf{iterate}'\,f^2\,(f^2\,x) && (3.8)\text{ and }\mathrm{EQ}^{\triangleright}_{\mathsf{next}} \\
&\qquad = \mathsf{iterate}'\ f^2\,x
\end{aligned}$$

One might wonder why we use $\mathsf{iterate}'$ here instead of the more general $\mathsf{iterate}$; 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 1.10.2. 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,

$$\mathcal{L}(f \circ g) = \mathcal{L}(f) \circ \mathcal{L}(g) \tag{3.9}$$

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 3.10 and functional extensionality, (3.9) 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 4 for an example.

(3) In Section 1.4 we claimed there is an isomorphism between the types $\blacksquare A + \blacksquare B$ and $\blacksquare(A + B)$, witnessed by the terms

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

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 $\blacksquare$:

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

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:

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

As a final remark of this section, we note that our main direction of further work beyond this paper has been to extend the $\mathsf{g}\lambda$-calculus with dependent types [11], as we will discuss further in Section 5.2. In this setting proofs take place inside the calculus, as with proof assistants such as Coq [33] and Agda [39]. 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.

4. 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* [43] in the $\mathsf{g}\lambda$-calculus. This hence allows us to reason about such functions in $L\mathsf{g}\lambda$, instead of via bisimulation arguments.

4.1. 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 $\mathsf{g}\lambda$-calculus, and how the program logic $L\mathsf{g}\lambda$ can be used to reason about them.

Definition 4.1. 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. [35]).

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 [43, 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 $\mathsf{g}\lambda$-calculus setting by, for example, considering a pair of mutually recursive stream functions as a function producing a pair of streams.

Example 4.2.

(1) Assuming we have constant zero of type $\mathbf{N}$, the constant stream zeros of Example 1.9.4 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 $\mathsf{g}\lambda$-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 3.12.2. Now by Lemma 3.9 we have

$$\mathsf{plus}^{\mathsf{g}} = \lambda s_1.\lambda s_2.(\mathsf{hd}^{\mathsf{g}}\, s_1 + \mathsf{hd}^{\mathsf{g}}\, s_2) :: ((\mathsf{next}\,\mathsf{plus}^{\mathsf{g}}) \circledast (\mathsf{tl}^{\mathsf{g}}\, s_1) \circledast (\mathsf{tl}^{\mathsf{g}}\, s_2)). \tag{4.1}$$

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 4.2.3. Given $s_1, s_2 : \mathsf{Str}\,\mathbf{N}$, we have

$$\begin{array}{rcll} \mathsf{hd}(\mathsf{plus}\, s_1 s_2) & = & \mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}(\mathsf{plus}\, s_1\, s_2) & \\ & = & \mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}(\mathcal{L}_2(\mathsf{plus}^{\mathsf{g}})\, s_1\, s_2) & \\ & = & \mathsf{hd}^{\mathsf{g}}(\mathsf{plus}^{\mathsf{g}}(\mathsf{unbox}\, s_1)(\mathsf{unbox}\, s_2)) & \text{(Example 3.12.2)} \\ & = & (\mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}\, s_1) + (\mathsf{hd}^{\mathsf{g}}\,\mathsf{unbox}\, s_2) & (4.1) \\ & = & (\mathsf{hd}\, s_1) + (\mathsf{hd}\, s_2) & \end{array}$$

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 3.9.

4.2. **From Behavioural Differential Equations to $\mathsf{g}\lambda$-Terms.** In the previous section we saw an example of a translation from a behavioural differential equation to a $\mathsf{g}\lambda$-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 $\mathsf{g}\lambda$-term[6]

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

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 $\mathsf{g}\lambda$ with function symbols in the signature Σ of (h_f, t_f). Using these it is straightforward to define a $\mathsf{g}\lambda$ 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.

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

$$\vec{x}, \vec{y} : \mathsf{Str}^{\mathsf{g}} A, \vec{z} : \blacktriangleright\mathsf{Str}^{\mathsf{g}} A, f : \blacktriangleright\left((\mathsf{Str}^{\mathsf{g}} A)^k \to \mathsf{Str}^{\mathsf{g}} A\right) \vdash t^{\mathsf{g}}_f : \blacktriangleright\mathsf{Str}^{\mathsf{g}} A$$

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

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

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

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

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

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

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

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

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

Analogously from a behavioural differential equation we define a $\mathsf{g}\lambda$ term Φ_f of type

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

where for the function symbols we take the lifted (as in Example 3.12.2) 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 2.19).

4.3. **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 [32]. 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 2.1.

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

$$\Pi_1 : \mathcal{S} \to \mathbf{Set} \qquad \Delta : \mathbf{Set} \to \mathcal{S} \qquad \Gamma : \mathcal{S} \to \mathbf{Set}$$

$$\Pi_1(X) = X(1) \qquad \Delta(a)(\alpha) = \begin{cases} 1 & \text{if } \alpha = 0 \\ a & \text{otherwise} \end{cases} \qquad \Gamma(X) = X(\omega)$$

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

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

$$(\blacktriangleright X)(\nu + 1) = X(\nu)$$
$$(\blacktriangleright X)(\alpha) = X(\alpha)$$

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 4.3. *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*

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

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

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

Proof.

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

□

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.

4.4. **Expressing Behavioural Differential Equations.** We first define two interpretations of behavioural differential equations (Definition 4.1); 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 4.2, 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 4.2.

Definition 4.4. 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 $\mathsf{Str}^{\mathsf{g}}(\Delta|A|)$ from Example 2.4.1. 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:

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

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

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

and is defined as:

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

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

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

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 4.5. 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 $[\![g]\!]_{\mathbf{Set}} = \Gamma[\![g]\!]_{\mathcal{S}} = ([\![g]\!]_{\mathcal{S}})_\omega$.

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

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

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

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

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

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

as follows:

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

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

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

Lemma 4.6. *For the above defined F and $\underline{F}$ we have*

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

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

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

and

$$\underline{F}(\mathsf{lim}(\phi)) = \underline{F}\,(\phi_\omega)$$

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

$$\underline{F}\,(\phi_\omega)\,(\vec{\sigma}) = F_\omega(\phi)_\omega(\vec{\sigma})$$

Recall that F is the exponential transpose of F', so $F_\omega(\phi)_\omega(\vec{\sigma}) = F'_\omega(\phi, \vec{\sigma})$. 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

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

and also

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

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 4.7. *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 $\mathsf{g}\lambda$-term Φ_f^{g} of type*

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

and a term Φ_f of type

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

(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 3.12.2.)

Let $f^{\mathsf{g}} = \mathsf{fix}\,\Phi_f^{\mathsf{g}}$ be the fixed point of Φ_f^{g}. Then $f = \mathcal{L}_k(\mathsf{box}\,f^{\mathsf{g}})$ 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 4.3 is the interpretation of the term Φ_f^{g} from Section 4.2. The inclusion of the morphism $\underline{F}$ in Lemma 4.3 is the denotation of the term Φ_f. Further, the inclusion (with Δ) of the fixed point constructed in Lemma 4.3 is the denotation of f.

Proposition 4.6 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 $\mathsf{g}\lambda$-calculus to define its solution.

5. Concluding Remarks

We have seen how the guarded lambda-calculus, or $\mathsf{g}\lambda$-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 $L\mathsf{g}\lambda$ for reasoning about $\mathsf{g}\lambda$-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.

5.1. Related Work.

Other Calculi with Later. Since Nakano's original paper [38] there have been a number of calculi presented that utilise the later modality. Many of these calculi are *causal* [2, 27–30, 40, 44], 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 [29], where programs should indeed be prevented from reacting to an event before it has occurred. We could similarly program in the fragment of the $\mathsf{g}\lambda$-calculus without $\blacksquare$ to retain this guarantee. We further note that the $\mathsf{g}\lambda$-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 [42, Section 9]), or the use of natural numbers to stratify typing judgments [29], or reduction [2].

Atkey and McBride's clock quantifiers [4] showed how to express *acausal* functions in a calculus with later. This was extended to dependent types by Møgelberg [36], with improvements made subsequently by Bizjak and Møgelberg [12]. However the conference version of this paper [13] 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 [4] (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 [4], and so we no longer have a conventional destructor for $\blacktriangleright$, so $\beta\eta$-equalities become more complex. Reiterating our remarks of Section 1.1 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 $\blacksquare$ modality seems the simpler choice, especially as it allows us to adapt previously published work on term calculus for the modal logic Intuitionistic S4 [5]. However in our work on extending guarded type theory to dependent types [11] 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 [5]. Subsequent work by Davies and Pfenning [19] 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 3.7, 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 [7, Section 5]. This category $\mathcal{M}$ was shown to provide semantics for Nakano's calculus by Birkedal et al. [8], as well as for a related calculus with later by Krishnaswami and Benton [29]. These works do not feature the $\blacksquare$ 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 $\mathbf{0}$ type. If the empty space is added then $\blacktriangleright$ becomes undefinable: either $\blacktriangleright 0$ 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 $\blacktriangleright$ in constant contexts.

Sized Types. The best developed type-based method for ensuring productivity are sized types, introduced by Hughes et al. in 1996 [22]. 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 $L\mathsf{g}\lambda$; 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. [3] and including, for example, the program logic iCAP [46]. 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) [26, 45] as a basis for program logics. The Coq formalisation of the topos of trees via 'forcing' of Jaber et al. [23] 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 [18, 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 [25] and Jeffrey [24], look somewhat like $\blacktriangleright$ and $\blacksquare$, but we as yet see no obvious formal links between these approaches.

5.2. **Further Work.**

Dependent Types. As discussed earlier, a major goal of this research is to extend the simply-typed $\mathsf{g}\lambda$-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 3 into the calculus itself. In Bizjak et al. [11] 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 [6] based on *cubical type theory* [16], 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 $\mathsf{g}\lambda$ ***Type- and Term-Formers.*** The examples in this paper make clear that programming in the $\mathsf{g}\lambda$-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 2.19 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 $\mathsf{g}\lambda$-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 $\mathsf{g}\lambda$-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.

References

[1] Andreas Abel, *MiniAgda: Integrating sized and dependent types*, Partiality and recursion in interactive theorem provers (PAR), 2010, pp. 14–28.

[2] Andreas Abel and Andrea Vezzosi, *A formalized proof of strong normalization for guarded recursive types*, Programming languages and systems (APLAS), 2014, pp. 140–158.

[3] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon, *A very modal model of a modern, major, general type system*, Principles of programming languages (POPL), 2007, pp. 109–122.

[4] Robert Atkey and Conor McBride, *Productive coprogramming with guarded recursion*, International conference on functional programming (ICFP), 2013, pp. 197–208.

[5] Gavin M. Bierman and Valeria C. V. de Paiva, *On an intuitionistic modal logic*, Studia Logica **65** (2000), no. 3, 383–416.

[6] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi, *Guarded cubical type theory: Path equality for guarded recursion*, 2016. Submitted.

[7] 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 **8** (2012), no. 4.

[8] Lars Birkedal, Jan Schwinghammer, and Kristian Støvring, *A metric model of lambda calculus with guarded recursion*, Fixed points in computer science (FICS), 2010, pp. 19–25.

[9] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg, *The category-theoretic solution of recursive metric-space equations*, Theoretical Computer Science **411** (2010), no. 47, 4102–4122.

[10] Aleš Bizjak, Lars Birkedal, and Marino Miculan, *A model of countable nondeterminism in guarded type theory*, Rewriting and typed lambda calculi (RTA-TLCA), 2014, pp. 108–123.
[11] Aleš Bizjak, Hans B. Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal, *Guarded dependent type theory with coinductive types*, Foundations of software science and computation structures (FoSSaCS), 2016, pp. 20–35.
[12] Aleš Bizjak and Rasmus E. Møgelberg, *A model of guarded recursion with clock synchronisation*, Mathematical foundations of programming semantics (MFPS), 2015, pp. 83–101.
[13] Ranald Clouston, Aleš Bizjak, Hans B. Grathwohl, and Lars Birkedal, *Programming and reasoning with guarded recursion for coinductive types*, Foundations of software science and computation structures (FoSSaCS), 2015, pp. 407–421.
[14] ———, *Programming and reasoning with guarded recursion for coinductive types*, arXiv:1501.02925 (2015).
[15] Ranald Clouston and Rajeev Goré, *Sequent calculus in the topos of trees*, Foundations of software science and computation structures (FoSSaCS), 2015, pp. 133–147.
[16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg, *Cubical type theory: a constructive interpretation of the univalence axiom*, 2016. Unpublished.
[17] Thierry Coquand, *Infinite objects in type theory*, Types for proofs and programs (TYPES), 1993, pp. 62–78.
[18] Nils A. Danielsson and Thorsten Altenkirch, *Subtyping, declaratively: An exercise in mixed induction and coinduction*, Mathematics of program construction (MPC), 2010, pp. 100–118.
[19] Rowan Davies and Frank Pfenning, *A modal analysis of staged computation*, Journal of the ACM **48** (2001), no. 3, 555–604.
[20] Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks, *Mix-automatic sequences*, 2013. Workshop on Combinatorics on Words, contributed talk.
[21] Jörg Endrullis, Dimitri Hendriks, and Martin Bodin, *Circular coinduction in Coq using bisimulation-up-to techniques*, Interactive theorem proving (ITP), 2013, pp. 354–369.
[22] John Hughes, Lars Pareto, and Amr Sabry, *Proving the correctness of reactive systems using sized types*, Principles of programming languages (POPL), 1996, pp. 410–423.
[23] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau, *Extending type theory with forcing*, Logic in computer science (lics), 2012, pp. 395–404.
[24] Alan Jeffrey, *LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs*, Programming languages meets program verification (PLPV), 2012, pp. 49–60.
[25] Wolfgang Jeltsch, *Towards a common categorical semantics for linear-time temporal logic and functional reactive programming*, Mathematical foundations of programming semantics (MFPS), 2012, pp. 229–242.
[26] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer, *Higher-order ghost state*, 2016. Submitted.
[27] Neelakantan R. Krishnaswami, *Higher-order functional reactive programming without spacetime leaks*, International conference on functional programming (ICFP), 2013, pp. 221–232.
[28] Neelakantan R. Krishnaswami and Nick Benton, *A semantic model for graphical user interfaces*, International conference on functional programming (ICFP), 2011, pp. 45–57.
[29] ———, *Ultrametric semantics of reactive programs*, Logic in computer science (LICS), 2011, pp. 257–266.
[30] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann, *Higher-order functional reactive programming in bounded space*, Principles of programming languages (POPL), 2012, pp. 45–58.
[31] Tadeusz Litak, *Constructive modalities with provability smack*, 2014. Author's cut, v. 2.03.
[32] Saunders Mac Lane and Ieke Moerdijk, *Sheaves in geometry and logic: A first introduction to topos theory*, Springer, 2012.
[33] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2004. Version 8.0.
[34] Conor McBride and Ross Paterson, *Applicative programming with effects*, Journal of Functional Programming **18** (2008), no. 1, 1–13.
[35] Stefan Milius, Lawrence S. Moss, and Daniel Schwencke, *Abstract GSOS rules and a modular treatment of recursive definitions*, Logical Methods in Computer Science **9** (2013), no. 3.

[36] Rasmus E. Møgelberg, *A type theory for productive coprogramming via guarded recursion*, Computer science logic and logic in computer science (CSL-LICS), 2014.
[37] Rasmus E. Møgelberg and Patrick Bahr, *Reduction semantics for guarded recursion*, 2016. Unpublished.
[38] Hiroshi Nakano, *A modality for recursion*, Logic in computer science (LICS), 2000, pp. 255–266.
[39] Ulf Norell, *Towards a practical programming language based on dependent type theory*, Ph.D. Thesis, 2007.
[40] François Pottier, *A typed store-passing translation for general references*, Principles of programming languages (POPL), 2011, pp. 147–158.
[41] Dag Prawitz, *Natural deduction: A proof-theoretical study*, Dover Publications, 1965.
[42] Reuben N. S. Rowe, *Semantic types for class-based objects*, Ph.D. Thesis, 2012.
[43] Jan J. M. M. Rutten, *Behavioural differential equations: a coinductive calculus of streams, automata, and power series*, Theoretical Computer Science **308** (2003), no. 1, 1–53.
[44] Paula G. Severi and Fer-Jan J. de Vries, *Pure type systems with corecursion on streams: from finite to infinitary normalisation*, International conference on functional programming (ICFP), 2012, pp. 141–152.
[45] Filip Sieczkowski, Aleš Bizjak, and Lars Birkedal, *ModuRes: A Coq library for modular reasoning about concurrent higher-order imperative programming languages*, Interactive theorem proving (ITP), 2015, pp. 375–390.
[46] Kasper Svendsen and Lars Birkedal, *Impredicative concurrent abstract predicates*, Programming languages and systems (ESOP), 2014, pp. 149–168.
[47] Sergei Tabachnikov, *Dragon curves revisited*, The Mathematical Intelligencer **1** (2014), no. 36, 13–17.