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
{ranald.clouston,abizjak,hbugge,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 demonstrate the expressiveness of the calculus by showing the definability of solutions to Rutten's behavioural differential equations. We introduce a program logic with Löb induction for reasoning about the contextual equivalence of programs.

1 Introduction

The problem of ensuring that functions on coinductive types are well-defined has prompted a wide variety of work into productivity checking, and rule formats for coalgebra. *Guarded recursion* [11] guarantees productivity and unique solutions by requiring that recursive calls be nested under a constructor, such as cons (written ::) for streams. This can sometimes be established by a simple syntactic check, as for the stream toggle and binary stream function interleave below:

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

Such syntactic checks, however, are often too blunt and exclude many valid definitions. For example the *regular paperfolding sequence*, the sequence of left and right turns (encoded as 1 and 0) generated by repeatedly folding a piece of paper in half, can be defined via the function interleave as follows [12]:

```
paperfolds = interleave toggle paperfolds
```

This definition is productive, but the putative definition below, which also applies interleave to two streams and so apparently is just as well-typed, is not:

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

A. Pitts (Ed.): FOSSACS 2015, LNCS 9034, pp. 407–421, 2015.
DOI: 10.1007/978-3-662-46678-0_26

This equation is satisfied by any stream whose *tail* is the regular paperfolding sequence, so lacks a unique solution. Unfortunately the syntactic productivity checker of the proof assistant Coq [13] will reject both definitions.

A more flexible approach, first suggested by Nakano [19], 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* of natural numbers are defined by the guarded recursive equation

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

asserting that stream heads are available now, but tails only later. The type of $\mathsf{interleave}$ will be $\mathsf{Str}^{\mathsf{g}} \to \blacktriangleright\mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}}$, 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 $\blacktriangleright\mathsf{Str}^{\mathsf{g}}$, rather than $\mathsf{Str}^{\mathsf{g}}$, 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

$$\mathsf{every2nd}\ (\mathsf{x} :: \mathsf{x'} :: \mathsf{xs}) = \mathsf{x} :: \mathsf{every2nd}\ \mathsf{xs}$$

cannot be typed because it violates *causality* [15]: elements of the result stream depend on deeper elements of the argument stream. In some settings, such as 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 controlling when clock quantifiers may be introduced. Moreover clock quantifiers transform guarded recursive types into first-class *coinductive* types, with guarded recursion defining the rule format for their manipulation.

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 the type-former, via the Curry-Howard isomorphism, as an *S4* modality, and hence base our operational semantics on the established typed calculi for intuitionistic S4 (IS4) of Bierman and de Paiva [5]. This is sufficient to capture all examples in the literature, which use only one clock; for examples that require multiple clocks we suggest extending our calculus to a *multimodal* logic.

In this paper we present the guarded λ-calculus, $\mathsf{g}\lambda$, extending the simply typed λ-calculus with coinductive and guarded recursive types. We define call-by-name operational semantics, which blocks non-termination via recursive definitions unfolding indefinitely. We define adequate denotational semantics in the topos of trees [6] and as a consequence prove normalisation. We introduce a program logic $L\mathsf{g}\lambda$ for reasoning about the denotations of $\mathsf{g}\lambda$-programs; given adequacy this permits proofs about the operational behaviour of terms. The logic is based on the internal logic of the topos of trees, with modalities $\triangleright, \Box$ on predicates, and Löb induction for reasoning about functions on both guarded recursive and coinductive types. We demonstrate the expressiveness of the calculus by showing the definability of solutions to Rutten's behavioural differential equations [21], and show that $L\mathsf{g}\lambda$ can be used to reason about them, as an alternative to standard bisimulation-based arguments.

We have implemented the $\mathsf{g}\lambda$-calculus in Agda, a process we found helpful when fine-tuning the design of our calculus. The implementation, with many examples, is available at `http://cs.au.dk/~hbugge/gl-agda.zip`.

2 Guarded λ-calculus

This section presents the guarded λ-calculus, written $\mathsf{g}\lambda$, its call-by-name operational semantics, and its types, then gives some examples.

Definition 2.1. $\mathsf{g}\lambda$-terms *are given by the grammar*

$$\begin{aligned} t ::= {} & x \mid \langle\rangle \mid \mathsf{zero} \mid \mathsf{succ}\, t \mid \langle t, t\rangle \mid \pi_d t \mid \lambda x.t \mid tt \mid \mathsf{fold}\, t \mid \mathsf{unfold}\, t \\ & \mid \mathsf{next}\, t \mid \mathsf{prev}\, \sigma.t \mid \mathsf{box}\, \sigma.t \mid \mathsf{unbox}\, t \mid t \circledast t \end{aligned}$$

where $d \in \{1,2\}$, x is a variable and $\sigma = [x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$, usually abbreviated $[\vec{x} \leftarrow \vec{t}]$, is a list of variables paired with 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 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. If furthermore t is closed we simply write* $\mathsf{prev}\, t$*. We will similarly write* $\mathsf{box}\, \iota.t$ *and* $\mathsf{box}\, t$*. We adopt the convention that* prev *and* box *have highest precedence.*

We may extend $\mathsf{g}\lambda$ with sums; for space reasons these appear only in the extended version of this paper [9].

Definition 2.2. *The* reduction rules *on closed* $\mathsf{g}\lambda$*-terms are*

$$\begin{aligned} \pi_d\langle t_1, t_2\rangle &\mapsto t_d & & (d \in \{1,2\}) \\ (\lambda x.t_1)t_2 &\mapsto t_1[t_2/x] & & \\ \mathsf{unfold}\,\mathsf{fold}\, t &\mapsto t & & \\ \mathsf{prev}[\vec{x} \leftarrow \vec{t}].t &\mapsto \mathsf{prev}\, t[\vec{t}/\vec{x}] & & (\vec{x} \text{ non-empty}) \\ \mathsf{prev}\,\mathsf{next}\, t &\mapsto t & & \\ \mathsf{unbox}(\mathsf{box}[\vec{x} \leftarrow \vec{t}].t) &\mapsto t[\vec{t}/\vec{x}] & & \\ \mathsf{next}\, t_1 \circledast \mathsf{next}\, t_2 &\mapsto \mathsf{next}(t_1 t_2) & & \end{aligned}$$

The rules above look like standard β-reduction, removing 'roundabouts' of introduction then elimination, with the exception of those regarding prev and next. An apparently more conventional β-rule for these term-formers would be

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

but where $\vec{x}$ is non-empty this would require us to reduce an open term to derive $\mathsf{next}\, t$. We take the view that reduction of open terms is undesirable within a call-by-name discipline, so first apply the substitution without eliminating prev.

The final rule 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 [16].

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

Definition 2.3. Values *are terms of the form*

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

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

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

$$E ::= \cdot \mid \mathsf{succ}\, E \mid \pi_d E \mid E t \mid \mathsf{unfold}\, E \mid \mathsf{prev}\, E \mid \mathsf{unbox}\, E \mid E \circledast t \mid v \circledast E$$

If we regard $\circledast$ as a variant of function application, it is surprising in a call-by-name setting to reduce on both its sides. 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 call-by-name reductions of the corresponding λ-calculus term with the novel connectives erased.

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

Lemma 2.6. *The call-by-name reduction relation* $\mapsto$ *is deterministic.*

Definition 2.7. $\mathsf{g}\lambda$-types *are defined inductively by the rules of Fig. 1.* ∇ *is a finite set of* type variables*. 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.*

$$\frac{}{\nabla, \alpha \vdash \alpha} \qquad \frac{}{\nabla \vdash \mathbf{1}} \qquad \frac{}{\nabla \vdash \mathbf{N}} \qquad \frac{\nabla \vdash A_1 \qquad \nabla \vdash A_2}{\nabla \vdash A_1 \times A_2} \qquad \frac{\nabla \vdash A_1 \qquad \nabla \vdash A_2}{\nabla \vdash A_1 \to A_2}$$

$$\frac{\nabla, \alpha \vdash A}{\nabla \vdash \mu\alpha.A}\ \alpha\ \mathsf{guarded\ in}\ A \qquad \frac{\nabla \vdash A}{\nabla \vdash \blacktriangleright A} \qquad \frac{\cdot \vdash A}{\nabla \vdash \blacksquare A}$$

Fig. 1. Type formation for the $\mathsf{g}\lambda$-calculus

Note the side condition on the μ type-former, and the prohibition on $\blacksquare A$ for open A, which can also be understood as a prohibition on applying $\mu\alpha$ to any α with $\blacksquare$ above it. The intuition for these restrictions is that unique fixed points exist only where the variable is displaced in time by a $\blacktriangleright$, but $\blacksquare$ cancels out this displacement by giving 'everything now'.

Definition 2.8. *The* typing judgments *are given in Fig. 2. There $d \in \{1, 2\}$, and the* typing contexts *Γ are finite sets of pairs $x : A$ where x is a variable and A a closed type. Closed types are* constant *if all occurrences of $\blacktriangleright$ are beneath an occurrence of $\blacksquare$ in their syntax tree.*

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

$$\frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \pi_d t : A_d} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

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

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

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

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

Fig. 2. Typing rules for the $\mathsf{g}\lambda$-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 we use only one clock in this work). Its use as a side-condition to $\blacksquare$-introduction in Fig. 2 recalls (but is more general than) the 'essentially modal' condition for natural deduction for IS4 of Prawitz [20]. 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 $\lambda n.\, \mathsf{box}\, \iota.n : \mathbf{N} \to \blacksquare\mathbf{N}$.

In examples prev usually appears in its syntactic sugar forms

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

and similarly for box; the more general form is nonetheless necessary because $(\mathsf{prev}\, \iota.t)[\vec{u}/\vec{x}] = \mathsf{prev}[\vec{x} \leftarrow \vec{u}].t$. Getting substitution right in this setting is somewhat delicate. For example our reduction rule $\mathsf{prev}[\vec{x} \leftarrow \vec{t}].t \mapsto \mathsf{prev}\, t[\vec{t}/\vec{x}]$ breaches subject reduction on open terms (but not for closed terms). See Bierman and de Paiva [5] for more discussion of substitution with respect to IS4.

Lemma 2.9 (Subject Reduction). *$\vdash t : A$ and $t \rightsquigarrow u$ implies $\vdash u : A$.*

Example 2.10. (i) The type of guarded recursive streams of natural numbers, $\mathsf{Str}^{\mathsf{g}}$, is defined as $\mu\alpha.\, \mathbf{N} \times \blacktriangleright\alpha$. These provide the setting for all examples below, but other definable types include infinite binary trees, as $\mu\alpha.\, \mathbf{N} \times \blacktriangleright\alpha \times \blacktriangleright\alpha$, and potentially infinite lists, as $\mu\alpha.\, \mathbf{1} + (\mathbf{N} \times \blacktriangleright\alpha)$.

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

$$:: \triangleq \lambda n.\lambda s.\, \mathsf{fold}\langle n, s\rangle : \mathbf{N} \to \blacktriangleright\mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}}$$
$$\mathsf{hd}^{\mathsf{g}} \triangleq \lambda s.\pi_1\, \mathsf{unfold}\, s : \mathsf{Str}^{\mathsf{g}} \to \mathbf{N} \qquad \mathsf{tl}^{\mathsf{g}} \triangleq \lambda s.\pi_2\, \mathsf{unfold}\, s :: \mathsf{Str}^{\mathsf{g}} \to \blacktriangleright\mathsf{Str}^{\mathsf{g}}$$

then use the $\circledast$ term-former for observations deeper into the stream:

$$\mathsf{2nd}^{\mathsf{g}} \triangleq \lambda s.(\mathsf{next}\, \mathsf{hd}^{\mathsf{g}}) \circledast (\mathsf{tl}^{\mathsf{g}}\, s) : \mathsf{Str}^{\mathsf{g}} \to \blacktriangleright \mathbf{N}$$
$$\mathsf{3rd}^{\mathsf{g}} \triangleq \lambda s.(\mathsf{next}\, \mathsf{2nd}^{\mathsf{g}}) \circledast (\mathsf{tl}^{\mathsf{g}}\, s) : \mathsf{Str}^{\mathsf{g}} \to \blacktriangleright\blacktriangleright \mathbf{N}\ \cdots$$

(iii) Following Abel and Vezzosi [2, Sec. 3.4] we may define a fixed point combinator fix with type $(\blacktriangleright A \to A) \to A$ for any A. We use this to define a stream by iteration of a function: $\mathsf{iterate}$ takes as arguments a natural number and a function, but the function is not used until the 'next' step of computation, so we may reflect this with our typing:

$$\mathsf{iterate} \triangleq \lambda f.\, \mathsf{fix}\, \lambda g.\lambda n.n :: (g \circledast (f \circledast \mathsf{next}\, n)) : \blacktriangleright(\mathbf{N} \to \mathbf{N}) \to \mathbf{N} \to \mathsf{Str}^{\mathsf{g}}$$

We may hence define the guarded stream of natural numbers

$$\mathsf{nats} \triangleq \mathsf{iterate}\, (\mathsf{next}\, \lambda n.\, \mathsf{succ}\, n)\, \mathsf{zero}\, .$$

(iv) With $\mathsf{interleave}$, following our discussion in the introduction, we again may reflect in our type that one of our arguments is not required until the next step, defining the term $\mathsf{interleave}$ as:

$$\mathsf{fix}\, \lambda g.\lambda s.\lambda t.(\mathsf{hd}^{\mathsf{g}}\, s) :: (g \circledast t \circledast \mathsf{next}(\mathsf{tl}^{\mathsf{g}}\, s)) : \mathsf{Str}^{\mathsf{g}} \to \blacktriangleright\mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}}$$

This typing decision is essential to define the paper folding stream:

$$\mathsf{toggle} \triangleq \mathsf{fix}\, \lambda s.(\mathsf{succ}\, \mathsf{zero}) :: (\mathsf{next}(\mathsf{zero} :: s))$$
$$\mathsf{paperfolds} \triangleq \mathsf{fix}\, \lambda s.\, \mathsf{interleave}\, \mathsf{toggle}\ s$$

Note that the unproductive definition with $\mathsf{interleave}\, s\, \mathsf{toggle}$ cannot be made to type check: informally, $s : \blacktriangleright\mathsf{Str}^{\mathsf{g}}$ cannot be converted into a $\mathsf{Str}^{\mathsf{g}}$ by prev, as it is in the scope of a variable s whose type $\mathsf{Str}^{\mathsf{g}}$ is not constant. To see a less artificial non-example, try to define a filter function on streams which eliminates elements that fail some boolean test.

(v) μ-types are in fact *unique* fixed points, so carry both final coalgebra and initial algebra structure. To see the latter, observe that we can define

$$\mathsf{foldr} \triangleq \mathsf{fix}\,\lambda g\lambda f.\lambda s.f\langle \mathsf{hd}^{\mathsf{g}}\, s, g \circledast \mathsf{next}\, f \circledast \mathsf{tl}^{\mathsf{g}}\, s\rangle : ((\mathbf{N} \times \blacktriangleright A) \to A) \to \mathsf{Str}^{\mathsf{g}} \to A$$

and hence for example $\mathsf{map}^{\mathsf{g}}\, h : \mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}}$ is $\mathsf{foldr}\,\lambda x.(h\pi_1 x) :: (\pi_2 x)$.

(vi) The $\blacksquare$ type-former lifts guarded recursive streams to coinductive streams, as we will make precise in Ex. 3.4. Let $\mathsf{Str} \triangleq \blacksquare\mathsf{Str}^{\mathsf{g}}$. We define $\mathsf{hd} : \mathsf{Str} \to \mathbf{N}$ and $\mathsf{tl} : \mathsf{Str} \to \mathsf{Str}$ by $\mathsf{hd} = \lambda s.\, \mathsf{hd}^{\mathsf{g}}(\mathsf{unbox}\, s)$ and $\mathsf{tl} = \lambda s.\, \mathsf{box}\, \iota.\, \mathsf{prev}\, \iota.\, \mathsf{tl}^{\mathsf{g}}(\mathsf{unbox}\, s)$, and hence define observations deep into streams whose results bear no trace of $\blacktriangleright$, for example $\mathsf{2nd} \triangleq \lambda s.\, \mathsf{hd}(\mathsf{tl}\, s) : \mathsf{Str} \to \mathbf{N}$.
In general boxed functions lift to functions on boxed types by

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

(vii) The more sophisticated acausal function $\mathsf{every2nd} : \mathsf{Str} \to \mathsf{Str}^{\mathsf{g}}$ is

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

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

3 Denotational Semantics and Normalisation

This section gives denotational semantics for $\mathsf{g}\lambda$-types and terms, as objects and arrows in the topos of trees [6], the presheaf category over the first infinite ordinal ω (we give a concrete definition below). These 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. Note that for space reasons many proofs, and some lemmas, appear only in the extended version of this paper [9].

Definition 3.1. *The* topos of trees $\mathcal{S}$ *has, as objects* X*, families of sets* X_1, X_2, ... *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* $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$.

$\mathcal{S}$ is a cartesian closed category with products defined pointwise. Its 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 3.2. – *The category of sets* **Set** *is a full subcategory of $\mathcal{S}$ via the functor $\Delta : \mathbf{Set} \to \mathcal{S}$ with $(\Delta Z)_i = Z$, $r_i^{\Delta Z} = id_Z$, and $(\Delta f)_i = f$. Objects in this subcategory are called* constant objects. *In particular the terminal object* 1 *of $\mathcal{S}$ is $\Delta\{*\}$ and the* natural numbers object *is $\Delta\mathbb{N}$;*

- *Δ is left adjoint to $hom_{\mathcal{S}}(1,\text{-})$; write $\blacksquare$ for $\Delta \circ hom_{\mathcal{S}}(1,\text{-}) : \mathcal{S} \to \mathcal{S}$.* $\mathsf{unbox} : \blacksquare \dot{\to} id_{\mathcal{S}}$ *is the counit of the resulting comonad. Concretely $\mathsf{unbox}_i(x) = x_i$, i.e. the i'th component of $x : 1 \to X$ applied to $*$;*
- *$\blacktriangleright : \mathcal{S} \to \mathcal{S}$ is defined by $(\blacktriangleright X)_1 = \{*\}$ and $(\blacktriangleright X)_{i+1} = X_i$, with $r_1^{\blacktriangleright X}$ defined uniquely and $r_{i+1}^{\blacktriangleright X} = r_i^X$. Its action on arrows $f : X \to Y$ is $(\blacktriangleright f)_1 = id_{\{*\}}$ and $(\blacktriangleright f)_{i+1} = f_i$. The natural transformation* $\mathsf{next} : id_{\mathcal{S}} \dot{\to} \blacktriangleright$ *has* next_1 *unique and* $\mathsf{next}_{i+1} = r_i^X$ *for any X.*

Definition 3.3. *We interpet 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. $[\![\alpha]\!](\vec{W}, X, Y) = Y$;*
- *$[\![\mathbf{1}]\!]$ and $[\![\mathbf{N}]\!]$ are the constant functors $\Delta\{*\}$ and $\Delta\mathbb{N}$ respectively;*
- *$[\![A_1 \times A_2]\!](\vec{W}) = [\![A_1]\!](\vec{W}) \times [\![A_2]\!](\vec{W})$ and likewise for $\mathcal{S}$-arrows;*
- *$[\![A_1 \to A_2]\!](\vec{W}) = [\![A_2]\!](\vec{W})^{[\![A_2]\!](\vec{W}')}$ 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. 3.2).*
- *$[\![\mu\alpha.A]\!](\vec{W}) = \mathsf{Fix}(F)$, 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 [6, Sec. 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.*

Example 3.4. $[\![\mathsf{Str}^{\mathsf{g}}]\!]_i = \mathbb{N}^i$, with projections as restriction functions, so is an object of *approximations* of streams – first the head, then the first two elements, and so forth. $[\![\mathsf{Str}]\!]_i = \mathbb{N}^\omega$ at all levels, so is the constant object of streams. 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 $\blacksquare\mu\alpha.A_F$ is the constant object of the carrier of the final coalgebra for F [18, Thm. 2].

Lemma 3.5. *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 3.6. *Closed constant types denote constant objects in $\mathcal{S}$.*

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

Definition 3.7. *We interpret typing contexts $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ 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 $\mathcal{S}$. Exponentials are not pointwise, so we give explicitly:

- *$[\![\lambda x.t]\!]_i(\gamma)_j$ maps $a \mapsto [\![\Gamma, x : A \vdash t : B]\!]_j(\gamma\restriction_j, a)$, where $\gamma\restriction_j$ is the result of applying restriction functions to $\gamma \in [\![\Gamma]\!]_i$ to get an element of $[\![\Gamma]\!]_j$;*
- *$[\![t_1 t_2]\!]_i(\gamma) = ([\![t_1]\!]_i(\gamma)_i) \circ [\![t_2]\!]_i(\gamma)$;*

$[\![\mathsf{fold}\, t]\!]$ and $[\![\mathsf{unfold}\, t]\!]$ are defined via composition with the isomorphisms of Lem. 3.5. $[\![\mathsf{next}\, t]\!]$ and $[\![\mathsf{unbox}\, t]\!]$ are defined by composition with the natural transformations introduced in Def. 3.2. The final three cases are

- *$[\![\mathsf{prev}[x_1 \leftarrow t_1, \ldots].t]\!]_i(\gamma) \triangleq [\![t]\!]_{i+1}([\![t_1]\!]_i(\gamma), \ldots)$, where $[\![t_1]\!]_i(\gamma) \in [\![A_1]\!]_i$ is also in $[\![A_1]\!]_{i+1}$ by Lem. 3.6;*
- *$[\![\mathsf{box}[x_1 \leftarrow t_1, \ldots].t]\!]_i(\gamma)_j = [\![t]\!]_j([\![t_1]\!]_i(\gamma), \ldots)$, again using Lem. 3.6;*
- *$[\![t_1 \circledast t_2]\!]_1$ is defined uniquely; $[\![t_1 \circledast t_2]\!]_{i+1}(\gamma) \triangleq ([\![t_1]\!]_{i+1}(\gamma)_i) \circ [\![t_2]\!]_{i+1}(\gamma)$.*

Lemma 3.8. *Given typed terms in context $x_1 : A_1, \ldots, x_m : A_m \vdash t : A$ and $\Gamma \vdash t_k : A_k$ for $1 \le k \le m$, $[\![t[\vec{t}/\vec{x}]]\!]_i(\gamma) = [\![t]\!]_i([\![t_1]\!]_i(\gamma), \ldots, [\![t_m]\!]_i(\gamma))$.*

Theorem 3.9 (Soundness). *If $t \rightsquigarrow u$ then $[\![t]\!] = [\![u]\!]$.*

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 [7], 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 3.10. *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* $A \in \{\alpha, \mathbf{0}, \mathbf{1}, \mathbf{N}\}$;
- $\mathsf{bd}(A \times B) = \mathsf{min}(\mathsf{bd}(A), \mathsf{bd}(B))$, *and similarly for* $\mathsf{bd}(A \to B)$;
- $\mathsf{bd}(\mu\alpha.A) = \mathsf{bd}(A)$, *and similarly for* $\mathsf{bd}(\blacktriangleright A)$;
- $\mathsf{bd}(\blacksquare A) = \mathsf{bd}(A) + 1$.

Lemma 3.11. *(i) α guarded in A implies* $\mathsf{us}(A[B/\alpha]) \le \mathsf{us}(A)$.
(ii) $\mathsf{bd}(B) \le \mathsf{bd}(A)$ *implies* $\mathsf{bd}(A[B/\alpha]) \le \mathsf{bd}(A)$

Definition 3.12. *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*

- $*R_i^{\mathbf{1}}t$ *iff* $t \rightsquigarrow \langle\rangle$*;*
- $nR_i^{\mathbf{N}}t$ *iff* $t \rightsquigarrow \mathsf{succ}^n\,\mathsf{zero}$*;*
- $(a_1,a_2)R_i^{A_1\times A_2}t$ *iff* $t \rightsquigarrow \langle t_1,t_2\rangle$ *and* $a_dR_i^{A_d}t_d$ *for* $d \in \{1,2\}$*;*
- $fR_i^{A\to B}t$ *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. Lem. 3.5);*
- $aR_i^{\blacktriangleright A}t$ *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*,* $a_jR_j^{A}u$*;*

This 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. Lem. 3.11.(ii) 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 Lem. 3.11.(i) for* μ*-types.*

Lemma 3.13 (Fundamental Lemma). *Take* $\Gamma = (x_1 : A_1, \ldots, x_m : A_m)$*,* $\Gamma \vdash t : A$*, and* $\vdash t_k : A_k$ *for* $1 \leq k \leq m$*. Then for all* i*, if* $a_kR_i^{A_k}t_k$ *for all* k*, then*

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

Theorem 3.14 (Adequacy and Normalisation).

(i) For all closed terms $\vdash t : A$ *it holds that* $[\![t]\!]_iR_i^{A}t$*;*
(ii) $[\![\vdash t : \mathbf{N}]\!]_i = n$ *implies* $t \rightsquigarrow \mathsf{succ}^n\,\mathsf{zero}$*;*
(iii) All closed typed terms evaluate to a value.

Proof. (i) specialises Lem. 3.13 to closed types. $(ii), (iii)$ hold by (i) and inspection of Def. 3.12.

Definition 3.15. *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* closing *contexts* C *of type* $\mathbf{N}$*, the terms* $C[t]$ *and* $C[u]$ *reduce to the same value.*

Corollary 3.16. $[\![t]\!] = [\![u]\!]$ *implies* $t \simeq_{\mathsf{ctx}} u$*.*

Proof. $[\![C[t]]\!] = [\![C[u]]\!]$ by compositionality of the denotational semantics . Then by Thm. 3.14.(ii) they reduce to the same value.

4 Logic for Guarded Lambda Calculus

This section presents our program logic $L\mathsf{g}\lambda$ for the guarded λ-calculus. The logic is an extension of the internal language of $\mathcal{S}$ [6,10]. Thus it extends multi-sorted intuitionistic higher-order logic with two propositional modalities $\triangleright$ and $\square$, pronounced later and always respectively. The term language of $L\mathsf{g}\lambda$ includes the terms of $\mathsf{g}\lambda$, and the types of $L\mathsf{g}\lambda$ include types definable in $\mathsf{g}\lambda$. We write Ω for the type of propositions, and also for the subobject classifier of $\mathcal{S}$.

The rules for *definitional equality* extend the usual $\beta\eta$-laws for functions and products with new equations for the new $\mathsf{g}\lambda$ constructs, listed in Fig. 3.

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

$$\frac{\Gamma_\blacksquare \vdash t : A \qquad \Gamma \vdash \vec{t} : \Gamma_\blacksquare}{\Gamma \vdash \mathsf{prev}[\vec{x} \leftarrow \vec{t}].(\mathsf{next}\,t) = t\,\big[\vec{t}/\vec{x}\big]} \qquad \frac{\Gamma_\blacksquare \vdash t : \blacktriangleright A \qquad \Gamma \vdash \vec{t} : \Gamma_\blacksquare}{\Gamma \vdash \mathsf{next}\,\big(\mathsf{prev}[\vec{x} \leftarrow \vec{t}].t\big) = t\,\big[\vec{t}/\vec{x}\big]}$$

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

Fig. 3. Additional equations. The context $\Gamma_\blacksquare$ is assumed constant.

Definition 4.1. *A type X is* total and inhabited *if the formula* $\mathrm{Total}\,(X) \equiv \forall x : \blacktriangleright X, \exists x' : X, \mathbf{next}(x') =_{\blacktriangleright X} x$ *is valid.*

All of the gλ-types defined in Sec. 2 are total and inhabited (see the extended version [9] for a proof using the semantics of the logic), but that is not the case when we include sum types as the empty type is not inhabited.

Corresponding to the modalities $\blacktriangleright$ and $\blacksquare$ on types, we have modalities $\triangleright$ and $\Box$ on formulas. The modality $\triangleright$ is used to express that a formula holds only "later", that is, after a time step. It is given by a function symbol $\triangleright : \Omega \to \Omega$. The $\Box$ modality is used to express that a formula holds for all time steps. Unlike the $\triangleright$ modality, $\Box$ on formulas does not arise from a function on Ω [8]. As with box, it is only well-behaved in constant contexts, so we will only allow $\Box$ in such contexts. The rules for $\triangleright$ and $\Box$ are listed in Fig. 4.

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

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

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

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

Fig. 4. Rules for $\triangleright$ and $\Box$. The judgement $\Gamma \mid \Xi \vdash \phi$ expresses that in typing context Γ, hypotheses in Ξ prove ϕ. The converse entailment in $\forall\triangleright$ and $\exists\triangleright$ rules holds if Y is *total and inhabited*. In all rules involving the $\Box$ the context Γ is assumed constant.

The $\triangleright$ modality can in fact be defined in terms of $\mathsf{lift} : \blacktriangleright\Omega \to \Omega$ (called *succ* by Birkedal et al [6]) as $\triangleright = \mathsf{lift} \circ \mathsf{next}$. The lift function will be useful since it allows us to define predicates over guarded types, such as predicates on $\mathsf{Str}^{\mathsf{g}}$.

The semantics of the logic is given in $\mathcal{S}$; terms are interpreted as morphisms of $\mathcal{S}$ and formulas are interpreted via the subobject classifier. We do not present the semantics here; except for the new terms of $\mathsf{g}\lambda$, whose semantics are defined in Sec. 3, the semantics are as in [6,8].

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

Proposition 4.2. *The formula* $\Box(\mathsf{unbox}\, x =_A \mathsf{unbox}\, y) \Rightarrow x =_{\blacksquare A} y$ *is valid.*

There exists a fixed-point combinator of type $(\blacktriangleright A \to A) \to A$ for all types A in the logic (not only those of in $\mathsf{g}\lambda$) [6, Thm. 2.4]; we also write fix for it.

Proposition 4.3. *For any 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* $u =_A \mathsf{fix}\, f$*.*

In particular this can be used for recursive definitions of predicates. For instance if $P : \mathbf{N} \to \Omega$ is a predicate on natural numbers we can define a predicate $P_{\mathsf{Str}^{\mathsf{g}}}$ on $\mathsf{Str}^{\mathsf{g}}$ expressing that P holds for all elements of the stream:

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

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

Theorem 4.4. *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 Γ to A. Adequacy (Cor. 3.16) then implies that t_1 and t_2 are contextually equivalent.

Example 4.5. We list some properties provable using the logic. Except for the first property all proof details are in the extended version [9].

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

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

Unfolding the definition of $\mathsf{map}^{\mathsf{g}}$ from Ex. 2.10(vi) and using β-rules and Prop. 4.3 we have $\mathsf{map}^{\mathsf{g}}\, f\, xs = f\,(\mathsf{hd}^{\mathsf{g}}\, xs){::}(\mathsf{next}(\mathsf{map}^{\mathsf{g}}\, f)\circledast(\mathsf{tl}^{\mathsf{g}}\, xs))$. Equality of functions is extensional so we have to prove

$$\Phi \triangleq \forall xs : \mathsf{Str}^{\mathsf{g}}, \mathsf{map}^{\mathsf{g}}\, f\,(\mathsf{map}^{\mathsf{g}}\, g\, xs) =_{\mathsf{Str}^{\mathsf{g}}} \mathsf{map}^{\mathsf{g}}(f \circ g)\, xs.$$

The proof is by Löb induction, so we assume $\triangleright\Phi$ and take $xs : \mathsf{Str}^{\mathsf{g}}$. Using the above property of $\mathsf{map}^{\mathsf{g}}$ we unfold $\mathsf{map}^{\mathsf{g}}\, f\,(\mathsf{map}^{\mathsf{g}}\, g\, xs)$ to

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

and we unfold $\mathsf{map}^{\mathsf{g}}(f \circ g)\, xs$ to $f\,(g\,(\mathsf{hd}^{\mathsf{g}}\, xs)) :: (\mathsf{next}(\mathsf{map}^{\mathsf{g}}(f \circ g)) \circledast \mathsf{tl}^{\mathsf{g}}\, xs)$. Since $\mathsf{Str}^{\mathsf{g}}$ is a total type there is a $xs' : \mathsf{Str}^{\mathsf{g}}$ such that $\mathsf{next}\, xs' = \mathsf{tl}^{\mathsf{g}}\, xs$. Using this and the rule for $\circledast$ we have

$$\mathsf{next}(\mathsf{map}^{\mathsf{g}}\, f) \circledast ((\mathsf{next}(\mathsf{map}^{\mathsf{g}}\, g)) \circledast \mathsf{tl}^{\mathsf{g}}\, xs) =_{\blacktriangleright\mathsf{Str}^{\mathsf{g}}} \mathsf{next}(\mathsf{map}^{\mathsf{g}}\, f(\mathsf{map}^{\mathsf{g}}\, g\, xs'))$$

and $\mathsf{next}(\mathsf{map}^{\mathsf{g}}(f \circ g)) \circledast \mathsf{tl}^{\mathsf{g}}\, xs =_{\blacktriangleright\mathsf{Str}^{\mathsf{g}}} \mathsf{next}(\mathsf{map}^{\mathsf{g}}(f \circ g)\, xs')$. From the induction hypothesis $\triangleright\Phi$ we have $\triangleright(\mathsf{map}^{\mathsf{g}}(f \circ g)\, xs' =_{\mathsf{Str}^{\mathsf{g}}} \mathsf{map}^{\mathsf{g}}\, f\,(\mathsf{map}^{\mathsf{g}}\, g\, xs'))$ and so rule $\textsc{eq}^{\triangleright}_{\mathsf{next}}$ concludes the proof.

(ii) We can also reason about acausal functions. For any $n : \mathbf{N}, f : \mathbf{N} \to \mathbf{N}$,

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

where f^2 is $\lambda m.f\,(f\,m)$. The proof again uses Löb induction.

(iii) Since our logic is higher-order we can state and prove very general properties, for instance the following general property of map

$$\begin{aligned}&\forall P, Q : (\mathbf{N} \to \Omega), \forall f : \mathbf{N} \to \mathbf{N}, (\forall x : \mathbf{N}, P(x) \Rightarrow Q(f(x)))\\ &\Rightarrow \forall xs : \mathsf{Str}^{\mathsf{g}}, P_{\mathsf{Str}^{\mathsf{g}}}(xs) \Rightarrow Q_{\mathsf{Str}^{\mathsf{g}}}(\mathsf{map}^{\mathsf{g}}\,f\,xs).\end{aligned}$$

The proof illustrates the use of the property $\mathsf{lift} \circ \mathsf{next} = \triangleright$.

(iv) Given a closed term (we can generalise to terms in constant contexts) f of type $A \to B$ we have $\mathsf{box}\,f$ of type $\blacksquare(A \to B)$. Define $\mathcal{L}(f) = \mathsf{lim}(\mathsf{box}\,f)$ of type $\blacksquare A \to \blacksquare B$. For any closed term $f : A \to B$ and $x : \blacksquare A$ we can then prove $\mathsf{unbox}(\mathcal{L}(f)\,x) =_B f\,(\mathsf{unbox}\,x)$. Then using Prop. 4.2 we can, for instance, prove $\mathcal{L}(f \circ g) = \mathcal{L}(f) \circ \mathcal{L}(g)$.
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 transfer equalities proved for functions on guarded types to functions on $\blacksquare$'d types; see Sec. 5 for an example.

5 Behavioural Differential Equations in $\mathbf{g\lambda}$

In this section we demonstrate the expressivity of our approach by showing how to construct solutions to behavioural differential equations [21] in $\mathsf{g}\lambda$, and how to reason about such functions in $L\mathsf{g}\lambda$, rather than with bisimulation as is more traditional. These ideas are best explained via a simple example.

Supposing addition $+ : \mathbf{N} \to \mathbf{N} \to \mathbf{N}$ is given, then pointwise addition of streams, plus, can be defined by the following behavioural differential equation

$$\mathsf{hd}(\mathsf{plus}\,\sigma_1\,\sigma_2) = \mathsf{hd}\,\sigma_1 + \mathsf{hd}\,\sigma_2 \qquad\qquad \mathsf{tl}(\mathsf{plus}\,\sigma_1\,\sigma_2) = \mathsf{plus}(\mathsf{tl}\,\sigma_1)\,(\mathsf{tl}\,\sigma_2).$$

To define the solution to this behavioural differential equation in $\mathsf{g}\lambda$, we first translate it to a function on guarded streams $\mathsf{plus}^{\mathsf{g}} : \mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}}$, as

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

then define $\mathsf{plus} : \mathsf{Str} \to \mathsf{Str} \to \mathsf{Str}$ by $\mathsf{plus} = \mathcal{L}_2(\mathsf{plus}^{\mathsf{g}})$. By Prop. 4.3 we have

$$\mathsf{plus}^{\mathsf{g}} = \lambda s_1.\lambda s_2.(\mathsf{hd}^{\mathsf{g}}\,s_1 + \mathsf{hd}^{\mathsf{g}}\,s_2) :: ((\mathsf{next}\,\mathsf{plus}^{\mathsf{g}}) \circledast (\mathsf{tl}^{\mathsf{g}}\,s_1) \circledast (\mathsf{tl}^{\mathsf{g}}\,s_2)). \qquad (1)$$

This definition of plus satisfies the specification given by the behavioural differential equation above. Let $\sigma_1, \sigma_2 : \mathsf{Str}$ and recall that $\mathsf{hd} = \mathsf{hd}^{\mathsf{g}} \circ \lambda s.\,\mathsf{unbox}\,s$. Then use Ex. 4.5.(iv) and equality (1) to get $\mathsf{hd}(\mathsf{plus}\,\sigma_1\sigma_2) = \mathsf{hd}\,\sigma_1 + \mathsf{hd}\,\sigma_2$.

For tl we proceed similarly, also using that $\mathsf{tl}^{\mathsf{g}}(\mathsf{unbox}\,\sigma) = \mathsf{next}(\mathsf{unbox}(\mathsf{tl}\,\sigma))$ which can be proved using the β-rule for box and the η-rule for next.

Since $\mathsf{plus}^{\mathsf{g}}$ is defined via guarded recursion we can reason about it with Löb induction, for example to prove that it is commutative. Ex. 4.5.(iv) and Prop. 4.2 then immediately give that plus on *coinductive* streams Str is commutative.

Once we have defined $\mathsf{plus}^{\mathsf{g}}$ we can use it when defining other functions on streams, for instance stream multiplication $\otimes$ which is specified by equations

$$\mathsf{hd}(\sigma_1 \otimes \sigma_2) = (\mathsf{hd}\,\sigma_1) \cdot (\mathsf{hd}\,\sigma_2) \quad \mathsf{tl}(\sigma_1 \otimes \sigma_2) = (\rho(\mathsf{hd}\,\sigma_1) \otimes (\mathsf{tl}\,\sigma_2)) \oplus ((\mathsf{tl}\,\sigma_1) \otimes \sigma_2)$$

where $\rho(n)$ is a stream with head n and tail a stream of zeros, and $\cdot$ is multiplication of natural numbers, and using $\oplus$ as infix notation for plus. We can define $\otimes^{\mathsf{g}} : \mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}}$ by $\otimes^{\mathsf{g}} \triangleq$

$$\begin{aligned}&\mathsf{fix}\,\lambda f.\lambda s_1.\lambda s_2.\,((\mathsf{hd}^{\mathsf{g}}\,s_1) \cdot (\mathsf{hd}^{\mathsf{g}}\,s_2)) :: \\ &\qquad\qquad (\mathsf{next}\,\mathsf{plus}^{\mathsf{g}} \circledast (f \circledast \mathsf{next}\,\iota^{\mathsf{g}}(\mathsf{hd}^{\mathsf{g}}\,s_1) \circledast \mathsf{tl}^{\mathsf{g}}\,s_2) \circledast (f \circledast \mathsf{tl}^{\mathsf{g}}\,s_1 \circledast \mathsf{next}\,s_2))\end{aligned}$$

then define $\otimes = \mathcal{L}_2\,(\otimes^{\mathsf{g}})$. It can be shown that the function $\otimes$ so defined satisfies the two defining equations above. Note that the guarded $\mathsf{plus}^{\mathsf{g}}$ is used to define $\otimes^{\mathsf{g}}$, so our approach is *modular* in the sense of [17].

The example above generalises, as we can show that any solution to a behavioural differential equation in **Set** can be obtained via guarded recursion together with $\mathcal{L}_k$. The formal statement is somewhat technical and can be found in the extended version [9].

6 Discussion

Following Nakano [19], the $\blacktriangleright$ modality has been used as type-former for a number of λ-calculi for guarded recursion. Nakano's calculus and some successors [15,22,2] permit only *causal* functions. The closest such work to ours is that of Abel and Vezzosi [2], but due to a lack of destructor for $\blacktriangleright$ their (strong) normalisation result relies on a somewhat artificial operational semantics where the number of nexts that can be reduced under is bounded by some fixed natural number.

Atkey and McBride's extension of such calculi to acausal functions [4] forms the basis of this paper. We build on their work by (aside from various minor changes such as eliminating the need to work modulo first-class type isomorphisms) introducing normalising operational semantics, an adequacy proof with respect to the topos of trees, and a program logic.

An alterative approach to type-based productivity guarantees are *sized types*, introduced by Hughes et al [14] and now extensively developed, for example integrated into a variant of System F_ω [1]. Our approach offers some advantages, such as adequate denotational semantics, and a notion of program proof without appeal to dependent types, but extensions with realistic language features (e.g. following Møgelberg [18]) clearly need to be investigated.

Acknowledgements. We gratefully acknowledge our discussions with Andreas Abel, Tadeusz Litak, Stefan Milius, Rasmus Møgelberg, Filip Sieczkowski, and Andrea Vezzosi, and the comments of the reviewers. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish

Council for Independent Research for the Natural Sciences (FNU). Aleš Bizjak is supported in part by a Microsoft Research PhD grant.

References

1. Abel, A., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: ICFP, pp. 185–196 (2013)
2. Abel, A., Vezzosi, A.: A formalized proof of strong normalization for guarded recursive types. In: APLAS, pp. 140–158 (2014)
3. Appel, A.W., Melliès, P.A., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: POPL, pp. 109–122 (2007)
4. Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: ICFP, pp. 197–208 (2013)
5. Bierman, G.M., de Paiva, V.C.: On an intuitionistic modal logic. Studia Logica 65(3), 383–416 (2000)
6. Birkedal, L., Møgelberg, R.E., Schwinghammer, J., Støvring, K.: First steps in synthetic guarded domain theory: step-indexing in the topos of trees. LMCS 8(4) (2012)
7. Birkedal, L., Schwinghammer, J., Støvring, K.: A metric model of lambda calculus with guarded recursion. In: FICS, pp. 19–25 (2010)
8. Bizjak, A., Birkedal, L., Miculan, M.: A model of countable nondeterminism in guarded type theory. In: Dowek, G. (ed.) RTA-TLCA 2014. LNCS, vol. 8560, pp. 108–123. Springer, Heidelberg (2014)
9. Clouston, R., Bizjak, A., Grathwohl, H.B., Birkedal, L.: Programming and reasoning with guarded recursion for coinductive types. arXiv:1501.02925 (2015)
10. Clouston, R., Goré, R.: Sequent calculus in the topos of trees. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 133–147. Springer, Heidelberg (2015)
11. Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994)
12. Endrullis, J., Grabmayer, C., Hendriks, D.: Mix-automatic sequences. In: Fields Workshop on Combinatorics on Words, contributed talk (2013)
13. Giménez, E.: Codifying guarded definitions with recursive schemes. In: Smith, J., Dybjer, P., Nordström, B. (eds.) TYPES 1994. LNCS, vol. 996, pp. 39–59. Springer, Heidelberg (1995)
14. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL, pp. 410–423 (1996)
15. Krishnaswami, N.R., Benton, N.: Ultrametric semantics of reactive programs. In: LICS, pp. 257–266 (2011)
16. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Programming 18(1), 1–13 (2008)
17. Milius, S., Moss, L.S., Schwencke, D.: Abstract GSOS rules and a modular treatment of recursive definitions. LMCS 9(3) (2013)
18. Møgelberg, R.E.: A type theory for productive coprogramming via guarded recursion. In: CSL-LICS (2014)
19. Nakano, H.: A modality for recursion. In: LICS, pp. 255–266 (2000)
20. Prawitz, D.: Natural Deduction: A Proof-Theoretical Study. Dover Publ. (1965)
21. Rutten, J.J.M.M.: Behavioural differential equations: A coinductive calculus of streams, automata, and power series. Theor. Comput. Sci. 308(1-3), 1–53 (2003)
22. Severi, P.G., de Vries, F.J.J.: Pure type systems with corecursion on streams: from finite to infinitary normalisation. In: ICFP, pp. 141–152 (2012)