A Light Modality for Recursion

Paula Severi

Department of Computer Science, University of Leicester, UK

Abstract. We investigate a modality for controlling the behaviour of recursive functional programs on infinite structures which is completely silent in the syntax. The latter means that programs do not contain "marks" showing the application of the introduction and elimination rules for the modality. This shifts the burden of controlling recursion from the programmer to the compiler.
To do this, we introduce a typed lambda calculus à la Curry with a silent modality and guarded recursive types. The typing discipline guarantees normalisation and can be transformed into an algorithm which infers the type of a program.

1 Introduction

The quest of finding a typing discipline that guarantees that functions on coinductive data types are productive has prompted a variety of works that rely on a modal operator [17,15,23,6,1,7,3]. All these works except for Nakano's [17] have explicit constructors and destructors in the syntax of programs [15,23,6,7]. This has the advantage that type inference is easy but it has the disadvantage that they do not really liberate the programmer from the task of controlling recursion since one has to know when to apply the introduction and elimination rules for the modal operator. As far as we know, decidability of type inference for Nakano's type system remains an open problem.

In this paper we consider the pure functional part of a type system studied previously [24]. This is a typed lambda calculus which has the advantage of having a modal operator silent in the syntax of programs without resorting to a subtyping relation as Nakano while keeping the nice properties of subject reduction and normalisation. We show that the type inference problem is decidable for the system under consideration. Even without a subtyping relation, this variant of the modal operator is still challenging to deal with because it is intrinsically non-structural, not corresponding to any expression form in the calculus.

Apart from the modal operator, we also include guarded recursive types which generalize the recursive equation $\mathtt{Str}_t = t \times \bullet\mathtt{Str}_t$ for streams [15,23]. This allows us to type productive functions on streams such as

$$\mathtt{skip}\, xs = \langle \mathtt{fst}\, xs, \mathtt{skip}\, (\mathtt{snd}\, (\mathtt{snd}\, xs))\rangle$$

which deletes the elements at even positions of a stream using the type $\mathtt{Str}_{\mathtt{Nat}} \to \mathtt{Str2}_{\mathtt{Nat}}$ where $\mathtt{Str2}_t = t \times \bullet\bullet\mathtt{Str2}_t$.

Lazy functional programming is acknowledged as a paradigm that fosters software modularity [12] and enables programmers to specify computations over possibly infinite data structures in elegant and concise ways. We give some examples that show how

modularization and compositionality can be achieved using the modal operator. An important recursive pattern used in functional programming for modularisation is $\texttt{foldr}$ defined by

$$\texttt{foldr}\ f\ xs = f\ (\texttt{fst}\ xs)\ (\texttt{foldr}\ f\ (\texttt{snd}\ xs))$$

The type of $\texttt{foldr}$ which is $(t \to \bullet s \to s) \to \texttt{Str}_t \to s$, is telling us what is the safe way to build functions. While it is possible to type

$$\texttt{iterate}\ f = \texttt{foldr}\ (\lambda xy.\langle f\ x, y\rangle)$$

by assigning the type $t \to t$ to f, and assuming $s = \texttt{Str}_t$, it is not possible to type the unproductive function $\texttt{foldr}\ (\lambda xy.y)$. This is because $\lambda xy.y$ does not have type $(t \to \bullet s \to s)$.

In spite of the fact that $\texttt{Str}_{\texttt{Nat}} \to \texttt{Str2}_{\texttt{Nat}}$ is the type of $\texttt{skip}$ with the minimal number of bullets, it does not give enough information to the programmer to know that the composition of $\texttt{skip}$ with itself is still typeable. In order to assist the user in the task of modularization, we need to infer a more general type for $\texttt{skip}$, which would look like

$$\texttt{Stream}(N,X) \to \texttt{Stream}(N+1,X)$$

where $\texttt{Stream}(N,X) = X \times \bullet^N \texttt{Stream}(N,X)$ and N and X are integer and type variable, respectively. It is clear now that from the above type, a programmer can deduce that it is possible to do the composition of $\texttt{skip}$ with itself.

Contributions and Outline. Section 2 defines the typed lambda calculus $\lambda^{\bullet}_{\to}$ with a silent modal operator and proves subject reduction and normalisation. Section 3 gives an adequate denotational semantics for $\lambda^{\bullet}_{\to}$ in the topos of trees as a way of linking our system to the work by Birkedal et al [2]. Section 4 shows the most important contribution of this paper which is decidability of the type inference problem for $\lambda^{\bullet}_{\to}$. This problem is solved by an algorithm which has the interesting feature of combining unification of types with integer linear programming. Sections 5 and 6 discuss related and future work, respectively.

2 A Light Modality for Typed Lambda Calculus

The syntax for *expressions* and *types* is given by the following grammars.

$e ::=^{ind}$		**Expression**	$t ::=^{coind}$		**Pseudo-type**
	x	(variable)		X	(type variable)
$\mid$	$\lambda x.e$	(abstraction)	$\mid$	$t \times t$	(product)
$\mid$	ee	(application)	$\mid$	$t \to t$	(arrow)
$\mid$	$\langle e, e\rangle$	(pair)	$\mid$	$\bullet t$	(delay)
$\mid$	$(\texttt{fst}\ e)$	(first)			
$\mid$	$(\texttt{snd}\ e)$	(second)			

In addition to the usual constructs of the λ-calculus, expressions include pairs. We do not need a primitive constant for the fixed point operator because it can be expressed

and typed inside the language. Expressions are subject to the usual conventions of the λ-calculus. In particular, we assume that the bodies of abstractions extend as much as possible to the right, that applications associate to the left, and we use parentheses to disambiguate the notation when necessary.

The syntax of *pseudo-types* is defined co-inductively. A type is a possibly infinite tree, where each internal node is labelled by a type constructor $\to$, $\times$ or $\bullet$ and has as many children as the arity of the constructor. The leaves of the tree (if any) are labelled by either type variables or $\mathtt{end}$. We use a co-inductive syntax to describe infinite data structures (such as streams). The syntax for pseudo-types include the types of the simply typed lambda calculus, *arrows* and *products* and the *delay* type constructor $\bullet$ [17]. An expression of type $\bullet t$ denotes a value of type t that is available "at the next moment in time". This constructor is key to control recursion and attain normalisation.

Definition 1 (Types). *We say that a pseudo-type t is*

1. regular *if its tree representation has finitely many distinct sub-trees.*
2. guarded *if every infinite path in its tree representation has infinitely many $\bullet$'s.*
3. *a* type *if it is regular and guarded.*

The regularity condition implies that we only consider types admitting a finite representation. It is equivalent to representing types with μ-notation and a strong equality which allows for an infinite number of unfoldings. This is also called the *equirecursive approach* since it views types as the unique solutions of recursive equations [10,19]. The existence and uniqueness of a solution satisfying condition 1 follow from known results (see [9] and also Theorem 7.5.34 of [5]). For example, there are unique types $\mathtt{Str}'_{\mathtt{Nat}}$, $\mathtt{Str}_{\mathtt{Nat}}$, and $\bullet^\infty$ that respectively satisfy the equations $\mathtt{Str}'_{\mathtt{Nat}} = \mathtt{Nat} \times \mathtt{Str}'_{\mathtt{Nat}}$, $\mathtt{Str}_{\mathtt{Nat}} = \mathtt{Nat} \times \bullet\mathtt{Str}_{\mathtt{Nat}}$, and $\bullet^\infty = \bullet\bullet^\infty$.

The guardedness condition intuitively means that not all parts of an infinite data structure can be available at once: those whose type is prefixed by a $\bullet$ are necessarily "delayed" in the sense that recursive calls on them must be deeper. For example, $\mathtt{Str}_{\mathtt{Nat}}$ is a type that denotes streams of natural numbers, where each subsequent element of the stream is delayed by one $\bullet$ compared to its predecessor. Instead $\mathtt{Str}'_{\mathtt{Nat}}$ is not a type: it would denote an infinite stream of natural numbers, whose elements are all available right away. If the types are written in μ-notation, the guardedness condition means that all occurrences of X in the body t of $\mu X.t$ are in the scope of a $\bullet$.

The type $\bullet^\infty$ is somehow degenerated in that it contains no actual data constructors. Unsurprisingly, we will see that non-normalising terms such as $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$ can only be typed with $\bullet^\infty$ (see Theorem 1). Without Condition 2, Ω could be given any type since the recursive pseudo-type $\mathtt{D} = \mathtt{D} \to t$ would become a type.

Sometimes we will write $\bullet^n t$ in place of $\underbrace{\bullet \cdots \bullet}_{n\text{-times}} t$.

We adopt the usual conventions regarding arrow types (which associate to the right) and assume the following precedence among type constructors: $\to$, $\times$, $\bullet$ with $\bullet$ having the highest precedence.

Expressions reduce according to a standard *call-by-name* semantics:

$$\text{[R-BETA]}\ (\lambda x.e_1)\ e_2 \longrightarrow e_1[e_2/x] \qquad \text{[R-FIRST]}\ \mathtt{fst}\ \langle e_1, e_2\rangle \longrightarrow e_1 \qquad \text{[R-SECOND]}\ \mathtt{snd}\ \langle e_1, e_2\rangle \longrightarrow e_2 \qquad \text{[R-CTXT]}\ \frac{e \longrightarrow f}{\mathscr{E}[e] \longrightarrow \mathscr{E}[f]}$$

where the *evaluation contexts* are $\mathscr{E} ::= [\,] \mid \mathscr{E}\,e \mid (\texttt{fst}\ \mathscr{E}) \mid (\texttt{snd}\ \mathscr{E})$. *Normal forms* are defined as usual as expressions that do not reduce.

The *type assignment system* $\lambda^{\bullet}_{\rightarrow}$ is defined by the following rules.

$$\frac{}{\Gamma, x : t \vdash x : t}\ [\text{AXIOM}] \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash e : \bullet t}\ [\bullet\text{I}] \qquad \frac{\Gamma, x : \bullet^n t \vdash e : \bullet^n s}{\Gamma \vdash \lambda x.e : \bullet^n (t \to s)}\ [\to\text{I}] \qquad \frac{\Gamma \vdash e_1 : \bullet^n (t \to s) \quad \Gamma \vdash e_2 : \bullet^n t}{\Gamma \vdash e_1 e_2 : \bullet^n s}\ [\to\text{E}]$$

$$\frac{\Gamma \vdash e_1 : \bullet^n t \qquad \Gamma \vdash e_2 : \bullet^n s}{\Gamma \vdash \langle e_1 , e_2 \rangle : \bullet^n (t \times s)}\ [\times\text{I}] \qquad \frac{}{\Gamma \vdash \texttt{fst} : t_1 \times t_2 \to t_1}\ [\times\text{E}_1] \qquad \frac{}{\Gamma \vdash \texttt{snd} : t_1 \times t_2 \to t_2}\ [\times\text{E}_2]$$

They are essentially the same as those for the simply typed lambda calculus except for two important differences. First, we now have an additional rule [•I] for introducing the modality. This rule is unusual in the sense that the expression remains the same, i.e. we do not have a constructor for • at the level of expressions. Second, each rule allows for an arbitrary delay in front of the types of the entities involved. Intuitively, the number of •'s represents the delay at which a value becomes available. So for example, rule [→I] says that a function which accepts an argument x of type t delayed by n and produces a result of type s delayed by the same n has type $\bullet^n(t \to s)$, that is a function delayed by n that maps elements of t into elements of s. Crucially, it is not possible to *anticipate* a delayed value: if it is known that a value will only be available with delay n, then it will also be available with any delay $m \geq n$, but not earlier.

Using rule [•I] and the recursive type $\texttt{D} = \bullet\texttt{D} \to t$, we can derive that the fixed point combinator $\texttt{fix} = \lambda y.(\lambda x.y\ (x\ x))(\lambda x.y\ (x\ x))$ has type $(\bullet t \to t) \to t$ by assigning the type $\texttt{D} \to t$ to the first occurrence of $\lambda x.y\ (x\ x)$ and $\bullet\texttt{D} \to t$ to the second one [17].

Let $\texttt{skip} = \texttt{fix}\ \lambda f x.\langle(\texttt{fst}\ x), f\ (\texttt{snd}\ (\texttt{snd}\ x))\rangle$ be the function that deletes the elements at even positions of a stream. In order to assign the type $\texttt{Str}_{\texttt{Nat}} \to \texttt{Str2}_{\texttt{Nat}}$ to skip, the variable f has to be delayed once and the first occurrence of snd has to be delayed twice. Note also that when typing the application $f\ (\texttt{snd}\ (\texttt{snd}\ x))$ the rule [→E] is used with $n = 2$.

The following lemma expresses the fact that the type of an expression should be delayed as much as the types in the environment. The proof is by induction on the derivation.

Lemma 1 (Delay). *If* $\Gamma \vdash e : t$, *then* $\Gamma_1, \bullet\Gamma_2 \vdash e : \bullet t$ *for* $\Gamma_1, \Gamma_2 = \Gamma$.

For example, from $x : t \vdash \lambda y.x : s \to t$ we can deduce that $x : \bullet t \vdash \lambda y.x : \bullet(s \to t)$, but we cannot deduce $x : \bullet t \vdash \lambda y.x : s \to t$. The Delay Lemma is crucial for proving Inversion Lemma:

Lemma 2 (Inversion).

1. *If* $\Gamma \vdash x : t$, *then* $t = \bullet^n t'$ *and* $x \in \mathsf{dom}(\Gamma)$.
2. *If* $\Gamma \vdash \lambda x.e : t$, *then* $t = \bullet^n(t_1 \to t_2)$ *and* $\Gamma, x : \bullet^n t_1 \vdash e : \bullet^n t_2$.
3. *If* $\Gamma \vdash e_1 e_2 : t$, *then* $t = \bullet^n t_2$ *and* $\Gamma \vdash e_2 : \bullet^n t_1$ *and* $\Gamma \vdash e_1 : \bullet^n(t_1 \to t_2)$.
4. *If* $\Gamma \vdash \texttt{fst} : t$, *then* $t = \bullet^n(t_1 \times t_2 \to t_1)$.
5. *If* $\Gamma \vdash \texttt{snd} : t$, *then* $t = \bullet^n(t_1 \times t_2 \to t_2)$.

Proof. By case analysis and induction on the derivation. We only show Item 2 which is interesting because we need to shift the environment in time and apply Lemma 1. A derivation of $\Gamma \vdash \lambda x.e : t$ ends with an application of either $[\to\text{I}]$ or $[\bullet\text{I}]$. For the former case, the proof is immediate. If the last applied rule is $[\bullet\text{I}]$, then $t = \bullet t'$ and $\Gamma \vdash \lambda x.e : t'$. By induction $t' = \bullet^n(t_1 \to t_2)$ and $\Gamma, x : \bullet^n t_1 \vdash e : \bullet^n t_2$. Hence $t = \bullet t' = \bullet^{n+1}(t_1 \to t_2)$ and by Lemma 1 $\Gamma, x : \bullet^{n+1} t_1 \vdash e : \bullet^{n+1} t_2$.

An important consequence of Inversion Lemma is Subject Reduction:

Lemma 3 (Subject Reduction). *If $\Gamma \vdash e : t$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : t$.*

Proof. By induction on the definition of $\longrightarrow$. We only do the case $(\lambda x.e_1)\ e_2 \longrightarrow e_1[e_2/x]$. Suppose $\Gamma \vdash (\lambda x.e_1)\ e_2 : t$. By Item 3 of Lemma 2

$$t = \bullet^n t_2 \qquad \Gamma \vdash e_2 : \bullet^n t_1 \qquad \Gamma \vdash (\lambda x.e_1) : \bullet^n(t_1 \to t_2)$$

It follows from Item 2 of Lemma 2 that $\Gamma, x : \bullet^n t_1 \vdash e_1 : \bullet^n t_2$. By applying Substitution Lemma (which follows by an easy induction on expressions), we deduce that $\Gamma \vdash e_1[e_2/x] : \bullet^n t_2$.

Neither Nakano's type system nor ours is closed under η-reduction. For example, $y : \bullet(t \to s) \vdash \lambda x.(y\ x) : (t \to \bullet s)$ but $y : \bullet(t \to s) \not\vdash y : (t \to \bullet s)$. The lack of subject reduction for η-reduction does not seem important in the context of lazy evaluation where programs are closed terms and only weak head normalised.

Theorem 1 (Normalisation). *[24,25] If $\Gamma \vdash e : t$ and $t \neq \bullet^\infty$, then e reduces (in zero or more steps) to a normal form.*

The proof of the above theorem follows from the fact that $\lambda^\bullet_\to$ is included in the type system of [24] and the latter is normalising [25]. Notice that there are normalising expressions that cannot be typed, for example $\lambda x.\Omega\mathbf{I}$, where $\Omega = (\lambda y.y\ y)(\lambda y.y\ y)$ and $\mathbf{I} = \lambda z.z$. In fact Ω has type $\bullet^\infty$ and by previous theorem it cannot have other types, and this implies that the application $\Omega\mathbf{I}$ has no type.

3 Denotational Semantics

This section gives a denotational semantics for $\lambda^\bullet_\to$ where types and expressions are interpreted as objects and morphisms in the *topos of trees* $\mathsf{Set}^{\omega^{op}}$ [2]. We give a self-contained description of this topos as a cartesian closed category for a reader familiar with λ-calculus. The *topos of trees* $\mathscr{S}$ has as objects A families of sets $A_1, A_2, \ldots$ indexed by positive integers, equipped with family of restrictions $r^A_i : A_{i+1} \to A_i$. Types will be interpreted as family of sets (not just sets). Intuitively the family represents better and better sets of approximants for the elements of that type. Arrows $f : A \to B$ are families of functions $f_i : A_i \to B_i$ obeying the naturality condition $f_i \circ r^A_i = r^B_i \circ f_{i+1}$.

$$\begin{array}{ccccccc} A_1 & \xleftarrow{r^A_1} & A_2 & \xleftarrow{r^A_2} & A_3 & & \ldots \\ {\scriptstyle f_1}\downarrow & & {\scriptstyle f_2}\downarrow & & {\scriptstyle f_3}\downarrow & & \\ B_1 & \xleftarrow[r^B_1]{} & B_2 & \xleftarrow[r^B_2]{} & B_3 & & \ldots \end{array}$$

We define $r^A_{ii} = id_A$ and $r^A_{ij} = r_j \circ \ldots \circ r_{i-1}$ for $1 \leq j < i$. Products are defined pointwise. Exponentials B^A have as components the sets:

$$(B^A)_i = \{(f_1, \ldots, f_i) \mid f_j : A_j \to B_j \text{ and } f_j \circ r^A_j = r^B_j \circ f_{j+1}\}$$

and as restrictions $r^{A \Rightarrow B}_i(f_1, \ldots f_{i+1}) = (f_1, \ldots, f_i)$. We define $\mathsf{eval} : B^A \times A \to B$ as $\mathsf{eval}_i((f_1, \ldots, f_i), a) = f_i(a)$ and $\mathsf{curry}(f) : C \to B^A$ for $f : C \times A \to B$ as

$$\mathsf{curry}(f)_i(c) = (g_1, \ldots, g_i)$$

where $g_j(a) = f_j(r^A_{ij}(c), a)$ for all $a \in A_j$ and $1 \leq j \leq i$. The functor $\blacktriangleright\colon \mathscr{S} \to \mathscr{S}$ is defined on objects as $(\blacktriangleright A)_1 = \{*\}$ and $(\blacktriangleright A)_{i+1} = A_i$ where $r^{\blacktriangleright A}_1 =!$ and $r^{\blacktriangleright A}_{i+1} = r^A_i$ and on arrows $(\blacktriangleright f)_1 = id_{\{*\}}$ and $(\blacktriangleright f)_{i+1} = f_i$. We write $\blacktriangleright^n$ for the n-times composition of $\blacktriangleright$.

The natural transformation $\mathsf{next}_A : A \to \blacktriangleright A$ is given by $(\mathsf{next}_A)_1 =!$ and $(\mathsf{next}_A)_{i+1} = r^A_i$ which can easily be extended to a natural transformation $\mathsf{next}^n_A : A \to \blacktriangleright^n A$. It is not difficult to see that there are isomorphisms $\theta : \blacktriangleright A \times \blacktriangleright B \to \blacktriangleright (A \times B)$ and $\xi : (\blacktriangleright B)^{(\blacktriangleright A)} \to \blacktriangleright (B^A)$ which are also natural. These can also be easily extended to isomorphisms $\theta^n : \blacktriangleright^n A \times \blacktriangleright^n B \to \blacktriangleright^n (A \times B)$ and $\xi^n : (\blacktriangleright^n B)^{(\blacktriangleright^n A)} \to \blacktriangleright^n (B^A)$.

A type t is interpreted as a functor $[\![t]\!] \in (\mathscr{S}^{op} \times \mathscr{S})^k \to \mathscr{S}$ by fixing a superset $X_1 \ldots X_k$ of its free variables. The mixed variance is a way of solving the problem of the contra-variance and the functoriality of $[\![t]\!]$ since variables can appear positively and negatively [7].

$$\begin{array}{ll}
[\![X]\!]\overrightarrow{(V,W)} & = W_j \text{ if } X = X_j \text{ for } 1 \leq j \leq k \\
[\![t \times s]\!]\overrightarrow{(V,W)} & = [\![t]\!]\overrightarrow{(V,W)} \times [\![s]\!]\overrightarrow{(V,W)} \\
[\![t \to s]\!]\overrightarrow{(V,W)} & = [\![s]\!]\overrightarrow{(V,W)}^{[\![t]\!]\overrightarrow{(W,V)}} \\
[\![\bullet t]\!] & = \blacktriangleright \circ [\![t]\!]
\end{array}$$

In order to justify that the interpretation is well-defined, it is necessary to view the above definition as indexed sets and we do induction on the pair $(i, rank(t))$ taking the lexicographic order where $rank(t)$ is defined by $rank(\bullet t) = 0$ and $rank(t \times s) = rank(t \to s) = max(rank(t), rank(s)) + 1$. By writing the indices explicitly for $[\![\bullet t]\!]$, we obtain

$$([\![\bullet t]\!]\overrightarrow{(V,W)})_1 = \{*\} \qquad ([\![\bullet t]\!]\overrightarrow{(V,W)})_{i+1} = ([\![t]\!]\overrightarrow{(V,W)})_i$$

where the index decreases. For $t \times s$ and $t \to s$, the interpretation at i is defined in terms of the interpretations of t and s at i, so the rank decreases.

Alternatively, if we represent types in μ-notation, the interpretation can be defined by induction on the type by adding the case:

$$\begin{array}{l}
[\![\mu X.t]\!]\overrightarrow{(V,W)} = \mathsf{Fix}(F) \text{ where } F(V_1, W_1) = [\![t]\!](\overrightarrow{(V,W)}, (V_1, W_1)) \\
\qquad\qquad\qquad\qquad \text{and } \mathsf{Fix}(F) \text{ is the unique } A \text{ such that } F(A,A) \cong A
\end{array}$$

The existence of the fixed point $\mathsf{Fix}(F)$ follows from [2, Section 4.5] since F is *locally contractive*.

As it is common in categorical semantics, the interpretation is not defined on lambda terms in isolation but on typing judgements. In order to define terms as morphisms, we

need the context and the type to specify their domain and co-domain. Typing contexts $\Gamma = x_1 : t_1, \ldots x_k : t_k$ are interpreted as $[\![t_1]\!] \times \ldots \times [\![t_k]\!]$. The interpretation of typed expressions $[\![\Gamma \vdash e : t]\!] : [\![\Gamma]\!] \to [\![t]\!]$ is defined by induction on e (using Inversion Lemma):

$$\begin{array}{ll}
[\![\Gamma \vdash x : \bullet^n t]\!] & = \mathsf{next}^n \circ \pi_j \text{ if } x = x_j \text{ and } t_j = t \text{ and } 1 \leq j \leq k \\
[\![\Gamma \vdash e_1 e_2 : \bullet^n s]\!] & = \mathsf{eval} \circ \langle (\xi^n)^{-1} \circ [\![\Gamma \vdash e_1 : \bullet^n (t \to s)]\!], [\![\Gamma \vdash e_2 : \bullet^n t]\!] \rangle \\
[\![\Gamma \vdash \lambda x.e : \bullet^n (t \to s)]\!] & = \xi^n \circ \mathsf{curry}([\![\Gamma, x : \bullet^n t \vdash e : \bullet^n s]\!]) \\
[\![\Gamma \vdash \langle e_1 , e_2 \rangle : \bullet^n (t \times s)]\!] & = (\theta^n) \circ \langle [\![\Gamma \vdash e_1 : \bullet^n t]\!], [\![\Gamma \vdash e_2 : \bullet^n s]\!] \rangle \\
[\![\Gamma \vdash (\mathtt{fst}\ e) : \bullet^n t]\!] & = \pi_1 \circ (\theta^n)^{-1} \circ [\![\Gamma \vdash e : \bullet^n (t \times s)]\!] \\
[\![\Gamma \vdash (\mathtt{snd}\ e) : \bullet^n t]\!] & = \pi_2 \circ (\theta^n)^{-1} \circ [\![\Gamma \vdash e : \bullet^n (t \times s)]\!]
\end{array}$$

Lemma 4 (Semantic Substitution).

$$[\![\Gamma, x : \bullet^n t \vdash e_1 : \bullet^n s]\!] \circ \langle id, [\![\Gamma \vdash e_2 : \bullet^n t]\!] \rangle = [\![\Gamma \vdash e_1[e_2/x] : \bullet^n s]\!]$$

Proof. This follows by induction on e_1. We only show the case $e_1 = x$. By Inversion Lemma $s = \bullet^m t$. Then,

$$[\![\Gamma, x : \bullet^n t \vdash e_1 : \bullet^{m+n} t]\!] \circ \langle id, [\![\Gamma \vdash e_2 : \bullet^n t]\!] \rangle = \mathsf{next}^m \circ [\![\Gamma \vdash e_2 : \bullet^n t]\!] = [\![\Gamma \vdash e_2 : \bullet^{n+m} t]\!]$$

The last equality follows from a semantic delay lemma.

Theorem 2 (Soundness). *If* $\Gamma \vdash e : t$ *and* $e \longrightarrow e'$ *then* $[\![\Gamma \vdash e : t]\!] = [\![\Gamma \vdash e' : t]\!]$.

Proof. We show the case of [R-BETA]. Let $v_1 = [\![\Gamma, x : \bullet^n t \vdash e_1 : \bullet^n s]\!]$ and $v_2 = [\![\Gamma \vdash e_2 : \bullet^n t]\!]$.

$$\begin{array}{rl}
[\![\Gamma \vdash (\lambda x.e_1) e_2 : \bullet^n s]\!] & = \mathsf{eval} \circ \langle (\xi^n)^{-1} \circ \xi^n \circ \mathsf{curry}(v_1), v_2 \rangle = v_1 \circ \langle id, v_2 \rangle \\
& = [\![\Gamma \vdash e_1[e_2/x] : \bullet^n s]\!] \text{ by Lemma 4}
\end{array}$$

The denotational semantics of $\lambda^{\bullet}_{\to}$ in the topos of trees can be generalised to $\mathsf{Set}^{A^{op}}$ for a set A equipped with a well-founded relation.

4 A Type Inference Algorithm

In this section, we define a type inference algorithm for $\lambda^{\bullet}_{\to}$. Apart from the usual complications that come from having no type declarations, the difficulty of finding an appropriate type inference algorithm for $\lambda^{\bullet}_{\to}$ is due to the fact that the expressions do not have a constructor and destructor for $\bullet$. We do not know which sub-expressions need to be delayed as illustrated by the type derivation of `fix` where the first occurrence of $(\lambda x.y\ (x\ x))$ has a different derivation from the second one since [•I] is applied in different places [17]. Even worse, in case a sub-expression has to be delayed, we do not know how many times needs to be delayed to be able to type the whole expression.

The type inference algorithm infers *meta-types* which are a generalization of types where the $\bullet$ can be exponentiated with integer expressions, e.g. $\bullet^{N_1} X \to \bullet^{N_2 - N_1} X$. The algorithm proceeds in several stages. The first stage generates a meta-type T_0 and a set $\mathscr{C}_0$ of meta-type constraints from a given closed expression e. Secondly, the unification algorithm transforms $\mathscr{C}_0$ into a set $\mathscr{C}$ of recursive equations and it simultaneously generates a set $\mathscr{E}$ of integer constraints to guarantee that the meta-types have non-negative exponents. Thirdly, a set $\mathsf{gC}(\mathscr{C})$ of integers constraints is computed to ensure that the solution is guarded. If $\mathscr{E} \cup \mathsf{gC}(\mathscr{C})$ is solvable then e is typable and its type is obtained by substituting T_0 by the solutions of $\mathscr{C}$ and $\mathscr{E} \cup \mathsf{gC}(\mathscr{C})$.

4.1 Meta-types

The syntax for *pseudo meta-types* is defined below. A pseudo meta-type can contain type variables and (non-negative) integer expressions with variables. In this syntax, $\bullet t$ is written as $\bullet^1 t$. We identify $\bullet^0 t$ with t and $\bullet^E \bullet^{E'} t$ with $\bullet^{E+E'} t$. We define a *meta-type* as a pseudo meta-type that is regular and guarded.

$E ::=^{ind}$		**Integer Expression**	$T ::=^{coind}$		**Pseudo Meta-type**
	N	(integer variable)		X	(type variable)
$\mid$	$\mathtt{n}$	(integer number)	$\mid$	$T \times T$	(product)
$\mid$	$E+E$	(addition)	$\mid$	$T \to T$	(arrow)
$\mid$	$E-E$	(substraction)	$\mid$	$\bullet^E T$	(delay)

Let τ be a finite mapping from type variables to meta-types, denoted as $\{X_1 \mapsto T_1, \ldots, X_n \mapsto T_n\}$, and let ρ be a finite mapping from integer variables to integer expressions, denoted as $\{N_1 \mapsto E_1, \ldots, N_m \mapsto E_m\}$. Let also $\sigma = \tau \cup \rho$. We define the substitution on a meta-types and an integer expression as follows.

$$
\begin{array}{llll}
N\rho & = E \text{ if } N \mapsto E \in \rho & X\sigma & = T \text{ if } X \mapsto T \in \tau \\
\mathtt{n}\rho & = \mathtt{n} & (T_1 \to T_2)\sigma & = T_1\sigma \to T_2\sigma \\
(E_1+E_2)\rho & = E_1\rho + E_2\rho & (T_1 \times T_2)\sigma & = T_1\sigma \times T_2\sigma \\
(E_1-E_2)\rho & = E_1\rho - E_2\rho & (\bullet^E T)\sigma & = \bullet^{E\rho} T\sigma
\end{array}
$$

We say that $\sigma = \tau \cup \rho$ is a *ground substitution* if ρ maps all integer variables into natural numbers. We say that T is *ground* if it contains no integer variables and all the exponents of $\bullet$ are natural numbers. We identify a ground type with the type obtained from replacing the type constructor $\bullet^n$ in the syntax of meta-types with n consecutive $\bullet$'s in the syntax of types.

Definition 2 (Constraints). *A* meta-type constraint *is an equation* $T \stackrel{?}{=} T'$ *between finite meta-types. An integer constraint is either* $E \stackrel{?}{=} E'$ *or* $E \stackrel{?}{\geq} E'$ *or* $E \stackrel{?}{<} E'$.

Moreover, $\sigma \models T \stackrel{?}{=} T'$ means that $T\sigma = T'\sigma$. Similarly, we define $\rho \models E \stackrel{?}{=} E'$, $\rho \models E \stackrel{?}{\geq} E'$ and $\rho \models E \stackrel{?}{<} E'$. This notation extends to a set $\mathscr{C}$ (or $\mathscr{E}$) of constraints in the obvious way.

We say that $\mathscr{C}$ is *substitutional* if $\mathscr{C} = \{X_1 \stackrel{?}{=} T_1, \ldots, X_n \stackrel{?}{=} T_n\}$ and all variables $X_1, \ldots, X_n$ are pairwise different. Since a substitutional $\mathscr{C}$ is a set of recursive equations where $T_1, \ldots, T_n$ are finite meta-types, there exists a unique solution $\tau_{\mathscr{C}}$ such that $\tau_{\mathscr{C}}(X_i)$ is regular for all $1 \leq i \leq n$ [9], [5, Theorem 7.5.34]. Note that the unicity of the solution of a set of recursive equations would not be guaranteed if we were following the iso-recursive approach which allows only for a finite number of unfoldings $\mu X.t = t\{X \mapsto \mu X.t\}$ [4,5].

We say that $\mathscr{E}$ *grounds* T if $T\rho$ is ground for all ground substitutions ρ such that $\rho \models \mathscr{E}$. For example, $N_1 \geq N_2$ grounds $(\bullet^{N_1 - N_2} X)$ but $N_1 \leq N_2$ does not.

4.2 Constraint Typing Rules

Table 1 defines the *constraint typing rules*. We assume that Δ only contains declarations of the form $x : X$. This is needed for the proof of Item 2 of Theorem 3. If instead of generating the fresh variable x in [→I] we directly put $\bullet^N X_1$ in the context, then we would not know what value to assign N in the case of $e = x$ unless we look at the rest of the type derivation. For example, consider $x : \bullet^5\texttt{Nat} \vdash x : \bullet^7\texttt{Nat}$. Then N should be assigned the value 3 and not 5 if later we derive that $\lambda x.x : \bullet^2(\bullet^3 X \to \bullet^5 X)$.

Note that given an expression e, it is always possible to derive Δ and $\mathscr{C}$ such that $\Delta \vdash e : T \mid \mathscr{C}$ and the set $\mathscr{C}$ contains only constraints between finite meta-types.

Theorem 3 (Correctness of Constraint Typing). *Let $\Delta \vdash e : T \mid \mathscr{C}$ and $\Gamma = x_1 : t_1, \ldots, x_m : t_m$ and $\Delta = x_1 : X_1, \ldots, x_m : X_m$.*

1. *Let $\Delta\sigma$ and $T\sigma$ be ground. If $\sigma \models \mathscr{C}$ then $\Delta\sigma \vdash e : T\sigma$.*
2. *If $\Gamma \vdash e : t$ then there exists a ground substitution $\sigma \supseteq \{X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\}$ such that $\sigma \models \mathscr{C}$ and $T\sigma = t$ and $\mathsf{dom}(\sigma) \setminus \{X_1, \ldots, X_m\}$ is the set of fresh variables in the derivation of $\Gamma \vdash e : t$.*

Proof. Items 1 and 2 follow by induction on e. For Item 1, we prove the case of the abstraction. It follows from induction hypothesis that $\Delta\sigma, x : T_1\sigma \vdash e : T_2\sigma$. Since $\sigma \models \{T_1 \stackrel{?}{=} \bullet^N X_1, T_2 \stackrel{?}{=} \bullet^N X_2\}$, it is obvious that we also have that $\Delta\sigma, x : \bullet^{N\sigma} X_1\sigma \vdash e : \bullet^{N\sigma} X_2\sigma$. We can, then, apply [→I] to conclude that the abstraction has type $\bullet^{N\sigma}(X_1\sigma \to X_2\sigma)$ from the context $\Delta\sigma$.

For Item 2 we prove a few cases. Suppose $e = x$. By Inversion Lemma, we have that $x = x_i$ and $t = \bullet^n t_i$ for some i. We define $\sigma = \{X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\} \cup \{N \mapsto n\}$. It is easy to see that $\Delta\sigma = \Gamma$ and $t = (\bullet^N X_i)\sigma$.

Table 1. Constraint Typing Rules

[AXIOM]
$$\frac{}{\Delta, x : X \vdash x : \bullet^N X \mid \emptyset}$$

[→I]
$$\frac{\Delta, x : X \vdash e : T \mid \mathscr{C} \qquad X, X_1, X_2, N \text{ are fresh}}{\Delta \vdash \lambda x.e : \bullet^N(X_1 \to X_2) \mid \mathscr{C} \cup \{X \stackrel{?}{=} \bullet^N X_1, T \stackrel{?}{=} \bullet^N X_2\}}$$

[→E]
$$\frac{\Delta \vdash e_1 : T_1 \mid \mathscr{C}_1 \qquad \Delta \vdash e_2 : T_2 \mid \mathscr{C}_2 \qquad X_1, X_2, N \text{ are fresh}}{\Delta \vdash e_1 e_2 : \bullet^N X_2 \mid \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{\bullet^N(X_1 \to X_2) \stackrel{?}{=} T_1, \bullet^N X_1 \stackrel{?}{=} T_2\}}$$

[×I]
$$\frac{\Delta \vdash e_1 : T_1 \mid \mathscr{C}_1 \qquad \Delta \vdash e_2 : T_2 \mid \mathscr{C}_2 \qquad X_1, X_2, N \text{ are fresh}}{\Delta \vdash \langle e_1 , e_2 \rangle : \bullet^N(X_1 \times X_2) \mid \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{\bullet^N X_1 \stackrel{?}{=} T_1, \bullet^N X_2 \stackrel{?}{=} T_2\}}$$

[×E$_1$]
$$\frac{N, X_1, X_2 \text{ are fresh}}{\Delta \vdash \texttt{fst} : \bullet^N(X_1 \times X_2 \to X_1) \mid \emptyset}$$

[×E$_2$]
$$\frac{N, X_1, X_2 \text{ are fresh}}{\Delta \vdash \texttt{snd} : \bullet^N(X_1 \times X_2 \to X_2) \mid \emptyset}$$

Suppose $e = e_1 e_2$. By Inversion Lemma, $t = \bullet^n t_2$ and $\Gamma \vdash e_2 : \bullet^n t_1$ and $\Gamma \vdash e_1 : \bullet^n(t_1 \to t_2)$. By induction hypotheses there are $\sigma_1, \sigma_2 \supseteq \{X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\}$ such that $\sigma_1 \models \mathscr{C}_1$ and $\sigma_2 \models \mathscr{C}_2$ and $T_1\sigma_1 = \bullet^n(t_1 \to t_2)$ and $T_1\sigma_2 = \bullet^n t_2$. Since, by induction hypotheses, $\mathsf{dom}(\sigma_1) \setminus \{X_1, \ldots, X_m\}$ is the set of fresh variables in the derivation of $\Gamma \vdash e_1 : t$ and $\mathsf{dom}(\sigma_2) \setminus \{X_1, \ldots, X_m\}$ is the set of fresh variables in the derivation of $\Gamma \vdash e_2 : t$, we have that $\sigma_1 \cup \sigma_2$ is a function. We can now define the substitution σ as $(\sigma_1 \cup \sigma_2)\{N \mapsto n\}\{X_1 \mapsto t_1\}\{X_2 \mapsto t_2\}$.

4.3 Unification Algorithm

The *unification algorithm* is defined in Table 2. Given a set $\mathscr{C}$ of constraints, it returns a set of pairs $(\mathscr{C}', \mathscr{E})$ such that $\mathscr{C}'$ is substitutional and $\mathscr{E}$ grounds the meta-types in $\mathscr{C}'$. The extra argument $\mathscr{D}$ keeps tracks of the "visited type constraints" and guarantees termination. The function unify does not return only one solution but a set of solutions. If $\bullet^N X \stackrel{?}{=} \bullet^{N'} X'$ then there are two ways of solving this equation: either $N \geq N'$ and $X' = \bullet^{N-N'} X$ or $N' > N$ and $X = \bullet^{N'-N} X'$. We use $\oplus$ for the disjoint union to guarantee that the type constraint that is being processed is removed from the set $\mathscr{C}$. The case $\mathscr{C}' \oplus \{\bullet^E X \stackrel{?}{=} \bullet^{E'} T\}$ assumes T is of the form $T_1 \,\mathsf{op}\, T_2$. It is clear that if $E' < E$, then the unification does not have a solution, e.g. $\bullet^{-2} X \stackrel{?}{=} T_1 \,\mathsf{op}\, T_2$. The case for $\mathscr{C}' \oplus \{T_1 \stackrel{?}{=} T_2, X \stackrel{?}{=} T_2\}$ is crucial for getting a substitutional set. By adding $T_1 \stackrel{?}{=} T_2$ and removing $X \stackrel{?}{=} T_2$ from the set $\mathscr{C}$, we obtain an equivalent set of constraints that "reduces" the set of constraints for X. Since $\mathscr{D}$ already contains $T_1 \stackrel{?}{=} T_2$, this constraint will not be added again avoiding non-termination. The unification algorithm for the simply typed lambda calculus solves the problem of termination in this case by reducing the number of variables, i.e. checks if $X \notin T_1$ and then, substitutes the variable x by T_1 in the remaining set of constraints. With recursive types, however, we do not perform the occur check and the number of variables may not decrease since the variable X may not disappear after substituting X by T (because X occurs in T). In order to decrease the number of variables, we could perhaps substitute X by $FIX\ (\lambda X.T_1)$ where FIX gives the solution of the recursive equation $X = T_1$ as a possible *infinite* tree. But the problem of guaranteeing termination would still be present if the meta-type constraints are allowed to be infinite since the size of the constraints may not decrease in some cases. We would also have a similar problem with termination if we use multi-equations and a rewrite relation instead of giving a function such as unify [20].

We start the algorithm by invoking $\mathsf{unify}(\mathscr{C}_0, \mathscr{C}_0)$ where the first and second argument are the same. In the remaining recursive calls, the second argument either remains the same or it is extended with the type constraint that has been just processed.

The size $|\ \mathscr{C}\ |$ of a set of constraints is the sum of the number of type variables and type constructors in the left hand side of the type constraints. Since the type constraints are finite, the size is always finite. We define

$$\mathsf{SubT}(\mathscr{C}_0) = \{T \mid S_1 \stackrel{?}{=} S_2 \in \mathscr{C}_0 \text{ and either } S_1 \text{ or } S_2 \text{ contains } T\}$$
$$\mathsf{SubC}(\mathscr{C}_0) = \{T_1 \stackrel{?}{=} T_2 \mid T_1, T_2 \in \mathsf{SubT}(\mathscr{C}_0)\}$$

Table 2. Unification Algorithm

$$
\begin{array}{l}
\mathsf{unify}(\mathscr{C},\mathscr{D}) = \text{if } \mathscr{C} \text{ is substitutional then } \{(\mathscr{C},\emptyset)\} \\
\qquad \text{else Case } \mathscr{C} \text{ of} \\
\qquad \mathscr{C}' \oplus \{T \stackrel{?}{=} T\} \Rightarrow \mathsf{unify}(\mathscr{C}',\mathscr{D}) \\
\qquad \mathscr{C}' \oplus \{T \stackrel{?}{=} X\} \Rightarrow \mathsf{unify}(\mathscr{C}' \cup \{X \stackrel{?}{=} T\}, \mathscr{D} \cup \{X \stackrel{?}{=} T\}) \\
\qquad \mathscr{C}' \oplus \{X \stackrel{?}{=} T_1, X \stackrel{?}{=} T_2\} \Rightarrow \\
\qquad\quad \text{if } T_1 \stackrel{?}{=} T_2 \in \mathscr{D} \text{ then } \mathsf{unify}(\mathscr{C}' \cup \{X \stackrel{?}{=} T_1\}, \mathscr{D}) \\
\qquad\quad \text{else } \mathsf{unify}(\mathscr{C}' \cup \{X \stackrel{?}{=} T_1, T_1 \stackrel{?}{=} T_2\}, \mathscr{D} \cup \{T_1 \stackrel{?}{=} T_2\}) \\
\qquad \mathscr{C}' \oplus \{\bullet^E X \stackrel{?}{=} \bullet^{E'} X'\} \Rightarrow \\
\qquad\quad \{(\mathscr{C}_0, \mathscr{E}_0 \cup \{E' \geq E\}) \mid (\mathscr{C}_0,\mathscr{E}_0) \in \mathsf{unify}(\mathscr{C}_1,\mathscr{D})\} \cup \\
\qquad\quad \{(\mathscr{C}_0, \mathscr{E}_0 \cup \{E > E'\}) \mid (\mathscr{C}_0,\mathscr{E}_0) \in \mathsf{unify}(\mathscr{C}_2,\mathscr{D})\} \\
\qquad\quad \text{where } \mathscr{C}_1 = \mathscr{C}' \cup \{X \stackrel{?}{=} \bullet^{E'-E} X'\} \text{ and } \mathscr{C}_2 = \mathscr{C}' \cup \{X' \stackrel{?}{=} \bullet^{E-E'} X\} \\
\qquad \mathscr{C} \oplus \{\bullet^E X \stackrel{?}{=} \bullet^{E'} T\} \Rightarrow \\
\qquad\quad \{(\mathscr{C}_0, \mathscr{E}_0 \cup \{E' \geq E\}) \mid (\mathscr{C}_0,\mathscr{E}_0) \in \mathsf{unify}(\mathscr{C}_1,\mathscr{D})\} \\
\qquad\quad \text{where } \mathscr{C}_1 = \mathscr{C} \cup \{X \stackrel{?}{=} \bullet^{E'-E} T\} \\
\qquad \mathscr{C} \oplus \{\bullet^{E'} T \stackrel{?}{=} \bullet^E X\} \Rightarrow \mathsf{unify}(\mathscr{C} \cup \{\bullet^E X \stackrel{?}{=} \bullet^{E'} T\}, \mathscr{D}) \\
\qquad \mathscr{C} \oplus \{\bullet^E (T_1 \ \mathsf{op}_1 \ T_2) \stackrel{?}{=} \bullet^{E'} (T_1' \ \mathsf{op}_2 \ T_2')\} \Rightarrow \\
\qquad\quad \text{if } \mathsf{op}_1 \neq \mathsf{op}_2 \text{ then fail} \\
\qquad\quad \text{else } \{(\mathscr{C}_0, \mathscr{E}_0 \cup \{E \stackrel{?}{=} E')\}) \mid (\mathscr{C}_0,\mathscr{E}_0) \in \mathsf{unify}(\mathscr{C}',\mathscr{D})\} \\
\qquad\quad \text{where } \mathscr{C}' = \mathscr{C} \cup \{T_1 \stackrel{?}{=} T_1', T_2 \stackrel{?}{=} T_2'\}
\end{array}
$$

In the evaluation of $\mathsf{unify}(\mathscr{C}_0,\mathscr{C}_0)$, the argument $\mathscr{D}$ of the recursive calls satisfies $\mathscr{C}_0 \subseteq \mathscr{D} \subseteq \mathsf{SubC}(\mathscr{C}_0)$.

Theorem 4 (Termination and Correctness of Unification). *Let* $\mathscr{D} \subseteq \mathsf{SubC}(\mathscr{C}_0)$.

1. $\mathsf{unify}(\mathscr{C},\mathscr{D})$ *terminates*
2. *If* $(\mathscr{C}_0,\mathscr{E}_0) \in \mathsf{unify}(\mathscr{C},\mathscr{D})$ *then* $\mathscr{C}_0$ *is substitutive.*
3. *If* $\sigma \models \mathscr{C}$ *then there exists* $(\mathscr{C}_0,\mathscr{E}_0) \in \mathsf{unify}(\mathscr{C},\mathscr{D})$ *such that* $\sigma \models \mathscr{C}_0$ *and* $\sigma \models \mathscr{E}_0$.
4. *If* $(\mathscr{C}_0,\mathscr{E}_0) \in \mathsf{unify}(\mathscr{C},\mathscr{D})$ *and* ρ *is a ground substitution such that* $\rho \models \mathscr{E}_0$ *then* $\rho \cup \tau_{\mathscr{C}_0} \models \mathscr{C}$ *and* $\rho(\tau_{\mathscr{C}_0}(X))$ *is ground for all* X.

Proof. In order to prove Item 1, observe that the second argument increases or remains the same in each recursive call. In the cases it remains the same, it is easy to see that $|\ \mathscr{C}\ |$ decreases. Since $\mathscr{D} \subseteq \mathsf{SubC}(\mathscr{C}_0)$, we have that

$$(|\ \mathsf{SubC}(\mathscr{C}_0)\ | - |\ \mathscr{D}\ |, |\ \mathscr{C}\ |)$$

decreases with each recursive call and hence, the unification algorithm terminates.

Items 2, 3 and 4 follow by induction on the number of recursive calls. For Item 3 we prove the case $\mathscr{C} = \mathscr{C}' \oplus \{\bullet^E (T_1 \ \mathsf{op} \ T_2) \stackrel{?}{=} \bullet^{E'} (T_1' \ \mathsf{op} \ T_2')\}$. It follows from $\sigma \models \mathscr{C}$

Table 3. Generation of Guard Constraints

Let $\mathscr{C} = \{X_1 \stackrel{?}{=} T_1, \ldots, X_n \stackrel{?}{=} T_n\}$ be substitutional. We compute the set of inequality constraints as follows.

$$\mathsf{gC}(\mathscr{C}) = \bigcup\{0 \stackrel{?}{<} E \mid E \in \mathsf{gE}(X_i, R_i) \;\; 1 \leq i \leq n\}$$

where $R_1, \ldots, R_n$ are obtained by performing substitutions:

$$S_1 = T_1,\ S_2 = T_2\{X_1 \mapsto S_1\},\ \ldots,\ S_n = T_n\{X_1 \mapsto S_1\}\ldots\{X_{n-1} \mapsto S_{n-1}\}$$
$$R_n = S_n,\ R_{n-1} = S_{n-1}\{X_n \mapsto S_n\},\ \ldots,\ R_1 = S_1\{X_2 \mapsto S_2, \ldots, X_n \mapsto S_n\}$$

and gE is defined as follows:

$$\begin{array}{ll} \mathsf{gE}(X, X) & = \{0\} \\ \mathsf{gE}(X, \bullet^E T) & = \{E + E' \mid E' \in \mathsf{gE}(T)\} \\ \mathsf{gE}(X, T_1 \ \mathtt{op}\ T_2) & = \mathsf{gE}(X, T_1) \cup \mathsf{gE}(X, T_2) \end{array}$$

that $\sigma \models \mathscr{C}' \cup \{T_1 \stackrel{?}{=} T_1', T_2 \stackrel{?}{=} T_2'\}$ and $\sigma \models E \stackrel{?}{=} E'$. By induction hypothesis there exists $(\mathscr{C}_0, \mathscr{E}_0) \in \mathsf{unify}(\mathscr{C}' \cup \{T_1 \stackrel{?}{=} T_1', T_2 \stackrel{?}{=} T_2'\}, \mathscr{D})$ such that $\sigma \models \mathscr{C}_0$ and $\sigma \models \mathscr{E}_0$. Then $(\mathscr{C}_0, \mathscr{E}_0 \cup \{E \stackrel{?}{=} E'\}) \in \mathsf{unify}(\mathscr{C}, \mathscr{D})$ and $\sigma \models \mathscr{E}_0 \cup \{E \stackrel{?}{=} E'\}$.

For Item 4 we prove the case $\mathscr{C} = \mathscr{C}' \oplus \{\bullet^E(T_1 \ \mathsf{op}\ T_2) \stackrel{?}{=} \bullet^{E'}(T_1' \ \mathsf{op}\ T_2')\}$. Suppose $(\mathscr{C}_0, \mathscr{E}_0) \in \mathsf{unify}(\mathscr{C}, \mathscr{D})$ and $\rho \models \mathscr{E}_0$. Then $\mathscr{E}_0 = \mathscr{E}_0' \cup \{E \stackrel{?}{=} E'\}$. By induction hypothesis $\tau_{\mathscr{C}_0} \cup \rho \models \mathscr{C}' \cup \{T_1 \stackrel{?}{=} T_1', T_2 \stackrel{?}{=} T_2'\}$. Since $\rho \models \{E \stackrel{?}{=} E'\}$, we have that $\tau_{\mathscr{C}_0} \cup \rho \models \mathscr{C}$.

The unification algorithm is exponential on the size of the input. If we are only interested in knowing if the program is typeable or not, then the complexity could be reduced to PSpace since it would be sufficient to store just one solution at a time.

4.4 Generation of Guard Constraints

Table 3 defines an algorithm for computing the set of integer constraints needed to enforce that the types are guarded. Intuitively, the function $\mathsf{gE}(X, T)$ adds up the exponents of the bullets from the root of T to wherever X occurs in T and gC forces the resulting integer expression to be greater than 0. However, just forcing $\mathsf{gE}(X, T)$ to be greater than 0 does not work because $\mathscr{C}$ is a set of mutually recursive equations. The simplest way to track the exponents along several recursive equations seems to perform substitutions in the spirit of Bekič law [14]. Suppose $\mathscr{C} = \{X_1 = T_1, X_2 = T_2\}$. The constraint $\mathsf{gE}(X_1, T_1) > 0$ is sufficient only if X_1 does not occur in T_2. If X_1 occurs in T_2, however, we should also be able to "see the recursive occurrences of X_1" coming from the second recursive equation. For this, we define S_i and R_i similarly to the way one calculates the solution in μ-notation from a set of recursive equations (see Theorem 8.1.1 in [5]). We can omit the μ's because they are not needed.

Since the solutions of the recursive equations $X = \bullet(X \to Y)$ and $X = \bullet X \to Y$ are different and both guarded, the integer constraint to guarantee guardedness for $X =$

Table 4. Type Inference Algorithm

Let e be a closed expression. The type inference algorithm $\mathsf{infer}(e)$ proceeds as follows.

1. It first calculates the meta-type T_0 and the set $\mathscr{C}_0$ of meta-type constraints such that $\emptyset \vdash e : T_0 \mid \mathscr{C}_0$ using the typing rules of Table 1.
2. Then $\mathsf{infer}(e)$ returns the set of pairs $(T_0\tau_{\mathscr{C}}, \mathscr{E})$ such that $(\mathscr{C}, \mathscr{E}) \in \mathsf{unify}(\mathscr{C}_0, \mathscr{C}_0)$ and the following two conditions hold:
   (a) $T_0\tau_{\mathscr{C}}$ is either $\bullet^E X$ or $\bullet^E(T_1 \to T_2)$ or $\bullet^E(T_1 \times T_2)$ and
   (b) the set $\mathscr{E} \cup \mathsf{gC}(\tau_{\mathscr{C}})$ has a solution of non-negative integers where gC is the set of constraints computed as in Table 3.

$\bullet^N(\bullet^M X \to Y)$ should be $N + M \geq 1$ which is more general than just $M \geq 1$. It gets more complicated if we have several mutual recursive equations because the recursive variable has to be tracked through several equations. For example, consider the set

$$X_1 \stackrel{?}{=} \bullet^{N_1}(\bullet^{N_2} X_1 \to \bullet^{N_3} X_2) \quad X_2 \stackrel{?}{=} \bullet^{M_1}(\bullet^{M_2} X_2 \to \bullet^{M_3} X_3) \quad X_3 \stackrel{?}{=} \bullet^{K_1}(\bullet^{K_2} X_1 \to \bullet^{K_3} X_3)$$

then, the set $\mathscr{E}$ of integer constraints to enforce that the types are guarded is:

$$\{N_1 + N_2 \geq 1, N_1 + N_3 + M_1 + M_3 + K_1 + K_2 \geq 1, M_1 + M_2 \geq 1, K_1 + K_3 \geq 1\}$$

Theorem 5 (Correctness of the Guard Constraint Generation). *Let* $\mathscr{C} = \{X_1 \stackrel{?}{=} T_1, \ldots, X_n \stackrel{?}{=} T_n\}$ *be substitutive and* $\tau_{\mathscr{C}} = \{X_1 \mapsto S_1, \ldots, X_n \mapsto S_n\}$ *and* $S_i\rho$ *be grounded. Then,* $S_i\rho$ *is guarded for all* $1 \leq i \leq n$ *iff* $\rho \models \mathsf{gC}(\mathscr{C})$.

Proof. Suppose $\rho \not\models \mathsf{gC}(\mathscr{C})$. Then, there is $E \in \mathsf{gE}(X_i, R_i)$ such that $\rho(E) = 0$ then the path from the root of $R_i\rho$ to X_i has no bullets. Since $S_i\rho = T_i(\rho \circ \tau_{\mathscr{C}}) = R_i(\rho \circ \tau_{\mathscr{C}'})$ where $\tau_{\mathscr{C}'} = \{X_1 \mapsto R_1, \ldots, X_n \mapsto R_n\}$, we have that there exists an infinite path in $S_i\rho$ that has no bullets at all. For the converse, suppose $\rho \models \mathsf{gC}(\mathscr{C})$. Then, an infinite path in $S_i\rho$ is an infinite path in $R_i(\rho \circ \tau_{\mathscr{C}'})$ and this path has an infinite number of bullets because the path from the root of $R_i\rho$ to X_i has a number $n = E\rho > 0$ of bullets.

4.5 Type Inference Algorithm

The *type inference algorithm* defined in Table 4 returns a finite set of meta-types that cover all the possible types of a closed expression e, i.e. any type of e is an instantiation of one of those meta-types. Item 2a checks that the type is different from $\bullet^\infty$. Here, we are using the fact that $t \neq \bullet^\infty$ if and only if t is either $\bullet^n X$ or $\bullet^n(t_1 \to t_2)$ or $\bullet^n(t_1 \times t_2)$. In order to check that $\mathscr{E} \cup \mathsf{gC}(\tau_{\mathscr{C}})$ has a solution of non-negative integers in Item 2b, we can use any algorithm for linear integer programming [18]. It is easy to modify this algorithm to give a set of minimal solutions by minimizing the sum of all integer variables. For instance, if the solution is a meta-type T satisfying the recursive equation $T = \bullet^{N_1}(\bullet^{N_2} T \to S)$ with $N_1 + N_2 \geq 1$ then we have two minimal solutions $N_1 = 1$, $N_2 = 0$ and $N_1 = 0$, $N_2 = 1$.

As an example, consider $\lambda x.x$. Then $\mathsf{infer}(\lambda x.x)$ yields only one solution $\bullet^N(X \to \bullet^M X)$ which it is actually its most general type.

Consider now $\lambda x.xx$. Then $\mathsf{infer}(\lambda x.xx)$ gives a set of two meta-types. The first meta-type is $\bullet^{N_1}(X_1 \to \bullet^{N_2-N_1}X_4)$ where X_1 should satisfy the recursive equation

$$X_1 \stackrel{?}{=} \bullet^{N_2-(N_1+N_3)}(\bullet^{N_1+N_4-N_2}X_1 \to X_4)$$

and N_1,N_2,N_3,N_4 should all be non-negative and satisfy $N_1+N_4 \geq N_2 \geq N_1+N_3$ and $N_4-N_3 \geq 1$. The second meta-type is $\bullet^{N_1}(\bullet^{N_2-(N_1+N_4)}X_3 \to \bullet^{N_2-N_1}X_4)$ where X_3 should satisfy the recursive equation

$$X_3 \stackrel{?}{=} \bullet^{N_4-N_3}(X_3 \to X_4)$$

and N_1,N_2,N_3,N_4 should all be non-negative and satisfy $N_2 > N_1+N_4$ and $N_4 \geq N_3$ and $N_4-N_3 \geq 1$. These two meta-types "cover all solutions" in the sense that any type of $\lambda x.xx$ is an instantiation of one of these two meta-types.

Theorem 6 (Correctness of Type Inference).

1. $\mathsf{infer}(e)$ *gives a finite set of pairs* $(T,\mathscr{E})$ *such that there exists at least one ground substitution* ρ *such that* $\rho \models \mathscr{E}$.
2. *If* $(T,\mathscr{E}) \in \mathsf{infer}(e)$ *then* $\vdash e : t$ *and* $T\rho = t \neq \bullet^\infty$ *for all ground substitutions* $\rho \models \mathscr{E}$.
3. *If* $\vdash e : t$ *and* $t \neq \bullet^\infty$ *then there exists* $(T,\mathscr{E}) \in \mathsf{infer}(e)$ *such that* $T\sigma = t$ *and* $\sigma \models \mathscr{E}$.

Proof. This follows from Theorems 3, 4 and 5.

Type checking for $\lambda^\bullet_\to$ (given an an expression e and a type t check if $\vdash e : t$) can easily be solved by inferring the (finite) set of meta-types for e and checking whether one of these meta-types unifies with t.

We could try to define an alternative type inference algorithm for $\lambda^\bullet_\to$ by re-using the one for $\lambda\mu$ [4,5]. However, this option does not make the problem simpler. Consider

$$\mathtt{skipRep}\ xs = \langle \mathtt{fst}\ xs, \langle \mathtt{fst}\ xs, \mathtt{skipRep}\ (\mathtt{snd}\ (\mathtt{snd}\ xs))\rangle\rangle$$

This function has type $\mathtt{Str}_{\mathtt{Nat}} \to \mathtt{SStr2}_{\mathtt{Nat}}$ where $\mathtt{SStr2}_{\mathtt{Nat}} = \mathtt{Nat} \times \mathtt{Nat} \times \bullet\bullet\mathtt{SStr2}_{\mathtt{Nat}}$. The type of this function without bullets is $\mathtt{Str}'_{\mathtt{Nat}} \to \mathtt{Str}'_{\mathtt{Nat}}$. How can we decorate $\mathtt{Str}'_{\mathtt{Nat}} \to \mathtt{Str}'_{\mathtt{Nat}}$ in order to obtain $\mathtt{Str}_{\mathtt{Nat}} \to \mathtt{SStr2}_{\mathtt{Nat}}$? Since a recursive type can be seen as an infinite tree with a finite number k of sub-trees, we could decorate each sub-tree with a certain number n_k of bullets. However, since $\mathtt{Str}_{\mathtt{Nat}}$ has only one sub-tree, it would mean that all sub-trees are decorated with the same number of bullets. In order to obtain $\mathtt{SStr2}_{\mathtt{Nat}}$, the domain $\mathtt{Str}'_{\mathtt{Nat}}$ of the function can be decorated by putting 1 bullet in all its subtrees. However, in order to obtain $\mathtt{SStr2}_{\mathtt{Nat}}$, the range $\mathtt{Str}'_{\mathtt{Nat}}$ of the function needs to be decorated by putting 0 bullets in some subtrees and 1 in others.

5 Related Work

This paper is an improvement over past typed lambda calculi with a temporal modal operator in two respects. Firstly, we do not need any subtyping relation as in [17] and

secondly programs are not cluttered with constructs for the introduction and elimination of individuals of type $\bullet$ as in [15,23,16,1,6,7,3]. The type system of [3] is designed having that denotational semantics in mind and it is syntactically more involved. Section 3 shows that our type system being syntactically simpler (and not designed having the semantics in mind) still admits the same semantics.

If we restrict to the recursive types $\mathtt{Str}_t = t \times \bullet\mathtt{Str}_t$ for infinite lists then, $\lambda^{\bullet}_{\to}$ is essentially the same as the type system of [15]. If only recursive types for lists are available, one cannot create other recursive types such as trees but having only one bullet also limits the amount of functions on streams we can type, e.g. `skip` is not typable in the type systems of [15,23].

Another type-based approach for ensuring productivity are sized types [13]. Type systems using size types do not always have neat properties: strong normalisation is gained by contracting the fixed point operator inside a case and they lack the property of subject reduction [22]. Another disadvantage of size types is that they do not include negative occurrences of the recursion variable [13] which are useful for some applications [26].

The proof assistant Coq does not ensure productivity through typing but by means of a syntactic guardedness condition (the recursive calls should be guarded by constructors) [11,8] which is somewhat restrictive since it rules out some interesting functions [23,7].

A sound but not complete type inference algorithm for Nakano's type system is presented in [21]. This means that the expressions typable by the algorithm are also typable in Nakano's system but the converse is not true. Though this algorithm is tractable, it is not clear to which type system it corresponds.

6 Conclusions and Future Work

The typeability problem (finding out if the program is typeable or not) is trivial in $\lambda\mu$ because all expressions are typeable using $\mu X.X \to X$. In $\lambda^{\bullet}_{\to}$, this problem turns out to be interesting because it gives us a way of filtering non-normalising programs when the type is not $\bullet^{\infty}$. It is also challenging because it involves the generation of integer constraints. The type inference algorithm presented in this paper does not give a unique principal type but a finite set of "principal" types. Since this (finite) set of types covers *all* possible types, it is possible to have modularity. Moreover, this algorithm can be easily extended to infer types for the processes of [24] since an initial process is the parallel composition of the main thread $x \Leftarrow e$ with an arbitrary number of $(\mathtt{server}\ \ a\ e_i)$ where e and e_i are closed expressions typable in $\lambda^{\bullet}_{\to}$ extended with `IO` and session types. As we mentioned at the end of Section 4.3, the typeability problem in $\lambda^{\bullet}_{\to}$ can be solved in PSpace but it still remains to see if this problem is PSpace-hard. In any case, it is important to refine this algorithm and find optimization techniques (heuristics, use of concurrency, etc) to make it practical.

It will be interesting to investigate the interaction of this variant of the modal operator with dependent types. As observed in Section 2, our type system is not closed under η-reduction. We leave the challenge of attaining a normalising and decidable type system closed under $\beta\eta$-reduction for future research.

References

1. R. Atkey and C. McBride. Productive Coprogramming with Guarded Recursion. In G. Morrisett and T. Uustalu, editors, *proceedings of ICFP'13*, pages 197–208. ACM, 2013.
2. L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-indexing in the Topos of Trees. *Logical Methods in Computer Science*, 8(4), 2012.
3. A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *Proceedings of FOSSACS 2016*, pages 20–35, 2016.
4. F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Inf. Comput.*, 92(1):48–80, 1991.
5. F. Cardone and M. Coppo. Recursive types. In H. Barendregt, W. Dekkers, and R. Statman, editors, *Lambda Calculus with Types*, Perspectives in Logic, pages 377–576. Cambridge, 2013.
6. A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair Reactive Programming. In S. Jagannathan and P. Sewell, editors, *proceedings of POPL'14*, pages 361–372. ACM Press, 2014.
7. R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. Programming and Reasoning with Guarded Recursion for Coinductive Types. In A. M. Pitts, editor, *proceedings of FoSSaCS'15*, volume 9034 of *LNCS*, pages 407–421. Springer, 2015.
8. T. Coquand. Infinite Objects in Type Theory. In H. Barendregt and T. Nipkow, editors, *proceedings of TYPES'93*, volume 806 of *LNCS*, pages 62–78. Springer, 1993.
9. B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.
10. V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.
11. E. Giménez and P. Casterán. A tutorial on [co-]inductive types in coq. Technical report, Inria, 1998.
12. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
13. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of POPL'96*, pages 410–423, 1996.
14. C. Jones, editor. *Programming Languages and Their Definition - Hans Bekic (1936-1982)*, LNCS 177. Springer, 1984.
15. N. R. Krishnaswami and N. Benton. Ultrametric Semantics of Reactive Programs. In M. Grohe, editor, *proceedings of LICS'11*, pages 257–266. IEEE, 2011.
16. N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order Functional Reactive Programming in Bounded Space. In *Proceedings of POPL'12*, pages 45–58. ACM Press, 2012.
17. H. Nakano. A Modality for Recursion. In M. Abadi, editor, *proceedings of LICS'00*, pages 255–266. IEEE, 2000.
18. C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization : algorithms and complexity*. Dover Publications, Mineola (N.Y.), 1998.
19. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
20. F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 389–489. MIT Press, 2005.
21. R. N. S. Rowe. *Semantic Types for Class-based Objects*. PhD thesis, Imperial College London, 2012.
22. J. L. Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *Procedings of LICS 2013*, pages 233–242, 2013.
23. P. Severi and F.-J. de Vries. Pure Type Systems with Corecursion on Streams: from Finite to Infinitary Normalisation. In P. Thiemann and R. B. Findler, editors, *proceedings of ICFP'12*, pages 141–152. ACM Press, 2012.

24. P. Severi, L. Padovani, E. Tuosto, and M. Dezani-Ciancaglini. On Sessions and Infinite Data. In A. L. Lafuente and J. Proença, editors, *Proceedings of COORDINATION'16*, volume 9686 of *LNCS*, pages 245–261. Springer, 2016.
25. P. Severi, L. Padovani, E. Tuosto, and M. Dezani-Ciancaglini. On sessions and infinite data. *CoRR*, abs/1610.06362, 2016.
26. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *Proceedings of ESOP 2014*, pages 149–168, 2014.