

Linearity and Iterator Types for Gödel's System \mathcal{T}

Sandra Alves · Maribel Fernández ·
Mário Florido · Ian Mackie

Received: date / Accepted: date

Abstract System $\mathcal{L}_{\mathcal{G}}$ is a linear λ -calculus with numbers and an iterator, which, although imposing linearity restrictions on terms, has all the computational power of Gödel's System \mathcal{T} . System $\mathcal{L}_{\mathcal{G}}$ owes its power to two features: the use of a closed reduction strategy (which permits the construction of an iterator on an open function, but only iterates the function after it becomes closed), and the use of a liberal typing rule for iterators based on iterative types. In this paper, we study these new types, and show how they relate to intersection types. We also give a sound and complete type reconstruction algorithm for System $\mathcal{L}_{\mathcal{G}}$.

Keywords Polymorphic iteration, linearity, intersection types

1 Introduction

New insights into linearity have led to the development of rich computational models; to support them, new strategies of reduction and new notions of types and typing rules have been introduced (see for instance [16, 17, 1, 27, 3]).

System $\mathcal{L}_{\mathcal{G}}$, as defined in [3], extends the linear λ -calculus with numbers, booleans, pairs, and an iterator, using a liberal typing rule for iterators based on *iterative types*. More precisely, in System $\mathcal{L}_{\mathcal{G}}$ it is possible to construct iterators where in some cases the iterated function is used with different types each time, so we have a form of polymorphic iteration [26].

Unlike previous linear versions of System \mathcal{T} , System $\mathcal{L}_{\mathcal{G}}$ permits building an iterator term with an open function, but uses a reduction strategy that will block such subterms until

S. Alves
University of Porto, Department of Computer Science & LIACC, R. do Campo Alegre 1021/1055, 4169-007,
Porto, Portugal

M. Fernández
King's College London, Department of Computer Science, Strand, London, WC2R 2LS U.K.

M. Florido
University of Porto, Department of Computer Science & LIACC, R. do Campo Alegre 1021/1055, 4169-007,
Porto, Portugal

I. Mackie
LIX, École Polytechnique, 91128 Palaiseau Cedex, France

the function becomes closed (thus preserving linearity). This reduction strategy, which we call *closed reduction*, has its roots in work by Girard on cut-elimination strategies [18], and was used to devise efficient evaluation strategies in the λ -calculus (see [15, 17]).

Although linear systems are known to be computationally weak, System $\mathcal{L}_{\mathcal{G}}$ has all the power of Gödel's System \mathcal{T} (see [3] for details of the encoding of System \mathcal{T} in System $\mathcal{L}_{\mathcal{G}}$). The use of closed reduction (or more precisely, the fact that using closed reduction a linear system can deal with more general classes of terms) is essential to obtain this result: in [5] two linear versions of System \mathcal{T} , with and without closed-reduction, are analysed; the first is strictly more powerful, it can represent Ackermann's function whereas the latter cannot.

It is a curious result that duplication can be encoded in a syntactically linear system when a particular strategy for reduction is used. The consequence of this result is that iterators can play the role of exponentials, offering the programmer a choice in how duplication is achieved. From one perspective, the non-linear aspect of a calculus is redundant in the presence of iterators, and so language designers need only offer a linear calculus with iterators. Moreover this issue is related to a problem, still open, in category theory, where the goal is to relate monoidal categories (where the internal language is the linear λ -calculus [32]) with cartesian closed categories (where the internal language is the simply typed λ -calculus [28, 29]) through the addition of a natural numbers object (represented in the internal language with an iterator). This was our main motivation for the addition of numbers and iterators to the linear λ -calculus, but the analysis of the expressive power of linear λ -calculi with iterators has other applications too. We are using this work as a first step in the analysis of linearity in Turing complete programming languages (PCF), and iterator types provide a form of discrete polymorphism which has applications in non-linear languages as well.

A distinctive feature of System $\mathcal{L}_{\mathcal{G}}$ is the use of iterative types, that give System $\mathcal{L}_{\mathcal{G}}$ a polymorphic flavour. Iterative types do not increase the computational power of the system, indeed, a version of System $\mathcal{L}_{\mathcal{G}}$ without iterative types (that is, with the standard typing rule for iterators) still has all the power of System \mathcal{T} when closed reduction is used [6]. However, the use of polymorphic iterators allows us to write more concise programs in System $\mathcal{L}_{\mathcal{G}}$ (see the discussion on polymorphic iterators in [26]) and preserves its good computational properties: we still have a confluent and strongly normalising system. To motivate the study of iterative types in System \mathcal{T} , consider a function map : $(A \rightarrow B) \rightarrow A^n \rightarrow B^n$, such that $\text{map } f(x_1, \dots, x_n) = (f(x_1), \dots, f(x_n))$. If one wants to define such a function for a pair then a possible solution is

$$\text{map} = \lambda f x. \langle f(\pi_1 x), f(\pi_2 x) \rangle$$

Note that a non-empty tuple of any size is represented using pairs

$$(x_1, \dots, x_n) = \langle x_1, \dots, \langle x_n, \text{true} \rangle \rangle$$

If one wants to write such a function for a tuple of size 7, then a similar solution will be

$$\lambda f x. \langle f(\pi_1 x), \langle f(\pi_2 x), \langle f(\pi_3 x), \langle f(\pi_4 x), \langle f(\pi_5 x), \langle f(\pi_6 x), \langle f(\pi_7 x), \text{true} \rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle$$

The size of this function will grow considerably, for tuples of higher size (even more in a linear system where arguments f, x cannot be shared, and therefore, must be explicitly duplicated before used, as we will see later). Using iteration we can define such a function for a tuple of size n in the following way¹:

$$\text{map} = \lambda f x. F(\pi_2(\text{iterate } n \langle x, \text{true} \rangle \lambda y. \langle \pi_2(\pi_1 y), \langle f(\pi_1(\pi_1 y)), \pi_2 y \rangle \rangle)).$$

¹ Since the iteration associates pairs to the left, the purpose of F is to associate pairs to the right: $F(\langle \langle \langle x_1, x_2 \rangle, x_3 \rangle, \text{true} \rangle) = \langle x_1, \langle x_2, \langle x_3, \text{true} \rangle \rangle \rangle$.

However, such a function cannot be written in System \mathcal{T} , because the iterated function takes a tuple of a different type at each iteration, but with types of the form $(A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_{n-1} \rightarrow A_n)$. Such types are what we call iterative types, and allow us to encode functions in a more concise and clear way.

In this paper we will focus on these new types, give a Curry-style type system for System $\mathcal{L}_{\mathcal{G}}$, and relate it to intersection type assignment systems. Intersection types were introduced by Coppo and Dezani in [11], and since then they have been used to characterise classes of terms with specific normalisation properties (see e.g. [36,9]), to define type systems with principal typings [25], to define models for the λ -calculus [10], etc. General intersection type assignment systems are undecidable, but several decidable restrictions have been defined (see for example [8,24,14]). Iterative types can be seen as a new decidable restriction of intersection types based on iteration. The type system of System $\mathcal{L}_{\mathcal{G}}$ is decidable: one of the main contributions of this paper is a type reconstruction algorithm for System $\mathcal{L}_{\mathcal{G}}$.

The rest of this paper is structured as follows. In the next section we recall Gödel's System \mathcal{T} . In Section 3 we define System $\mathcal{L}_{\mathcal{G}}$: we give its syntax, reduction rules and a type system using iterative types. We also prove Subject Reduction, confluence and strong normalisation of System $\mathcal{L}_{\mathcal{G}}$ in this section. Section 4 gives a type reconstruction algorithm, including the iterator types, with soundness and completeness proofs. Section 5 contains a detailed analysis of iterator types. Section 6 concludes the paper.

This paper is a revised and extended version of [3,4]. For more detailed proofs and examples we refer to [2].

2 Background

Gödel's System \mathcal{T} is the simply typed λ -calculus (with arrow types and products, and the usual β -reduction and projection rules) where two basic types have been added: numbers (built from 0 and S; we write $S^n 0$ for $S \dots (S 0)$) and booleans (true and false) with a

recursor and a conditional defined by the reduction rules:

$$\begin{array}{ll} \text{rec } 0 \ u \ v \longrightarrow u & \text{cond true } u \ v \longrightarrow u \\ \text{rec } (S \ t) \ u \ v \longrightarrow v (\text{rec } t \ u \ v) \ t & \text{cond false } u \ v \longrightarrow v \end{array}$$

Figure 1 shows the entire system, using the so-called multiplicative presentation of the type rules which is equivalent to the usual additive one in the presence of Weakening and Contraction (see [16]). A typing context Γ is a list of assumptions of the form $x : A$, such that for any types A, B , if $x : A$ and $x : B$ appear in Γ , then $A = B$. System \mathcal{T} is confluent, strongly normalising and reduction preserves types [19].

Our first step towards building a linear version of System \mathcal{T} will be to replace the recursor by a simpler iterator:

$$\text{iter } 0 \ u \ v \longrightarrow u \qquad \text{iter } (S \ t) \ u \ v \longrightarrow v(\text{iter } t \ u \ v)$$

with the following typing rule:

$$\frac{\Gamma \vdash_{\mathcal{T}} t : \text{Nat} \quad \Theta \vdash_{\mathcal{T}} u : A \quad \Delta \vdash_{\mathcal{T}} v : A \rightarrow A}{\Gamma, \Theta, \Delta \vdash_{\mathcal{T}} \text{iter } t \ u \ v : A}$$

The iterator has the same computational power as the recursor. We refer to [19] for more details on this result, which is based on the following argument:

Axiom and Structural Rules:

$$\frac{}{x : A \vdash_{\mathcal{T}} x : A} \text{ (Axiom)} \quad \frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathcal{T}} t : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathcal{T}} t : C} \text{ (Exchange)}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : C}{\Gamma, x : A \vdash_{\mathcal{T}} t : C} \text{ (Weakening)} \quad \frac{\Gamma, x : A, x : A \vdash_{\mathcal{T}} t : C}{\Gamma, x : A \vdash_{\mathcal{T}} t : C} \text{ (Contraction)}$$

Logical Rules:

$$\frac{\Gamma, x : A \vdash_{\mathcal{T}} t : B}{\Gamma \vdash_{\mathcal{T}} \lambda x. t : A \rightarrow B} (\rightarrow \text{Intro}) \quad \frac{\Gamma \vdash_{\mathcal{T}} t : A \rightarrow B \quad \Delta \vdash_{\mathcal{T}} u : A}{\Gamma, \Delta \vdash_{\mathcal{T}} tu : B} (\rightarrow \text{Elim})$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : A \quad \Delta \vdash_{\mathcal{T}} u : B}{\Gamma, \Delta \vdash_{\mathcal{T}} \langle t, u \rangle : A \times B} (\times \text{Intro})$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : A \times B}{\Gamma \vdash_{\mathcal{T}} \pi_1(t) : A} (\times \text{Elim}) \quad \frac{\Gamma \vdash_{\mathcal{T}} t : A \times B}{\Gamma \vdash_{\mathcal{T}} \pi_2(t) : B} (\times \text{Elim})$$

Numbers:

$$\frac{}{\vdash_{\mathcal{T}} 0 : \text{Nat}} \text{ (Zero)} \quad \frac{\Gamma \vdash_{\mathcal{T}} t : \text{Nat}}{\Gamma \vdash_{\mathcal{T}} S t : \text{Nat}} \text{ (Succ)}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : \text{Nat} \quad \Theta \vdash_{\mathcal{T}} u : A \quad \Delta \vdash_{\mathcal{T}} v : A \rightarrow \text{Nat} \rightarrow A}{\Gamma, \Theta, \Delta \vdash_{\mathcal{T}} \text{rec } t \ u \ v : A} \text{ (Rec)}$$

Booleans:

$$\frac{}{\vdash_{\mathcal{T}} \text{true} : \text{Bool}} \text{ (True)} \quad \frac{}{\vdash_{\mathcal{T}} \text{false} : \text{Bool}} \text{ (False)}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} t : \text{Bool} \quad \Theta \vdash_{\mathcal{T}} u : A \quad \Delta \vdash_{\mathcal{T}} v : A}{\Gamma, \Theta, \Delta \vdash_{\mathcal{T}} \text{cond } t \ u \ v : A} \text{ (Cond)}$$

Fig. 1 System \mathcal{T}

1. An iterator can be encoded with a recursor, preserving types and reductions. To show this, define $\text{iter } t \ u \ v \stackrel{\text{def}}{=} \text{rec } t \ u \ (\lambda xy. vx)$, then the result follows by an induction on t .
2. A recursor can be encoded with an iterator, preserving types and reductions. To show this, define:

$$\text{rec } t \ u \ v \stackrel{\text{def}}{=} \pi_1(\text{iter } t \ \langle u, 0 \rangle (\lambda x. \langle v(\pi_1 x)(\pi_2 x), S(\pi_2 x) \rangle))$$

then the result follows by induction on t .

In this paper, when we refer to System \mathcal{T} it will be the system with iterators rather than recursors (it is also confluent, strongly normalising, and type preserving).

2.1 Polymorphic iteration: System $\mathcal{T}_{\mathcal{I}}$

We now introduce the notion of polymorphic iteration in System \mathcal{T} . We start by defining a set of *iterative types*.

Definition 1 Let A_0, \dots, A_n be a (non-empty) list of System \mathcal{T} types. $It(A_0, \dots, A_n)$ denotes a non-empty set of *iterative types* defined by induction on n :

$$\begin{aligned} n = 0 : It(A_0) &= \{A_0 \rightarrow A_0\} \\ n = 1 : It(A_0, A_1) &= \{A_0 \rightarrow A_1\} \\ n \geq 2 : It(A_0, \dots, A_n) &= It(A_0, \dots, A_{n-1}) \cup \{A_{n-1} \rightarrow A_n\} \end{aligned}$$

Note that $It(A_0) = It(A_0, A_0) = It(A_0, \dots, A_0)$.

Below we will write $\Delta \vdash_{\mathcal{T}_g} v : It(A_0, \dots, A_n)$ as an abbreviation for $\Delta \vdash_{\mathcal{T}_g} v : B$ for each $B \in It(A_0, \dots, A_n)$.

Now, consider a version of System \mathcal{T} where we replace the usual typing rule for the iterator by:

$$\frac{\Gamma \vdash_{\mathcal{T}_g} t : \text{Nat} \quad \Theta \vdash_{\mathcal{T}_g} u : A_0 \quad \Delta \vdash_{\mathcal{T}_g} v : It(A_0, \dots, A_n) \quad (\star)}{\Gamma, \Theta, \Delta \vdash_{\mathcal{T}_g} \text{iter } t \ u \ v : A_n} \text{ (Iter)}$$

(\star) where $n = m$ if $t \equiv S^m 0$, otherwise $n = 0$.

We call this system \mathcal{T}_g .

With this new typing rule, terms of the form $\text{iter } t \ u \ v$ are typed differently depending on whether we know or not the number of times that the function v should be iterated. If t is a number $S^n 0$ then we require v to be a function that can be iterated n times on u . Otherwise, if t is not (yet) a number, we require v to have a type that allows it to be iterated any number of times (i.e. u has type A and $v : A \rightarrow A$, for some type A).

System \mathcal{T}_g is more expressive than System \mathcal{T} , in the sense that we can write more compact programs, but we can trivially encode functions in System \mathcal{T}_g using functions in System \mathcal{T} , by replacing each iterator of the form $\text{iter } S^m 0 \ u \ v$, by the expanded term $v^m(u)$.

The same happens in other standard polymorphic type systems. For example with the let-style polymorphism of Hindley-Milner type system one can write more compact programs in the sense that they are equivalent to inlining the let-bound definitions and typing the resulting term with simple types (see [38, 26]).

We will now present some examples to motivate the use of polymorphic iteration. For that, we start by defining arrays of size n , with $n > 0$, as:

$$[x_1, \dots, x_n] = \langle x_1, \dots, \langle x_n, \text{true} \rangle \rangle$$

we write $[A]^n$ to denote the type $A \times \underbrace{(\dots (A \times \text{Bool}) \dots)}_n$.

Example 1 Consider the function *iterate*, such that, for each n :

$$\text{iterate } f \ u = [f^n(u), \dots, f(u), u]$$

This function behaves like the Haskell [23] program `reverse(take n (iterate f u))`. We encode *iterate* in System \mathcal{T}_g as:

$$\text{iterate} = \lambda f u. \text{iter } S^n 0 \ \langle u, \text{true} \rangle \ \lambda l. \langle f(\pi_1 \ l), l \rangle$$

We can easily prove by induction the correctness of this encoding. The resulting term is typable in System \mathcal{T}_g :

$$\Gamma, \Delta \vdash_{\mathcal{T}_g} \text{iterate} : (A \rightarrow A) \rightarrow A \rightarrow [A]^{n+1}$$

since

- $\Gamma, u : A \vdash_{\mathcal{T}_g} \langle u, \text{true} \rangle : [A]^1$, and
- $\Delta, f : A \rightarrow A \vdash_{\mathcal{T}_g} \lambda l. \langle f(\pi_1 l), l \rangle : [A]^1 \rightarrow [A]^2$,
- ...
- $\Delta, f : A \rightarrow A \vdash_{\mathcal{T}_g} \lambda l. \langle f(\pi_1 l), l \rangle : [A]^n \rightarrow [A]^{n+1}$

Example 2 Consider the function `foldr`, such that

$$\text{foldr } g \ b \ [u_1, \dots, u_n] = g \ u_1 \ (\dots (g \ u_n \ b) \dots)$$

We encode `foldr` in System \mathcal{L}_g as:

$$\text{foldr} = \lambda g b x. \pi_1(\text{iter } S^n 0 \ \langle b, x \rangle \ \lambda l. \langle g(\pi_1 l) (\pi_1(\pi_2 l)), \pi_2(\pi_2 l) \rangle)$$

The resulting term is typable in System \mathcal{T}_g :

$$\Gamma, \Delta \vdash_{\mathcal{T}_g} \text{foldr} : (A \rightarrow B \rightarrow B) \rightarrow B \rightarrow [A]^n \rightarrow B$$

since

- $\Gamma, b : B, x : [A]^n \vdash_{\mathcal{T}_g} \langle b, x \rangle : B \times [A]^n$, and
- $\Delta, g : A \rightarrow B \rightarrow B \vdash_{\mathcal{T}_g} \lambda l. \langle g(\pi_1 l) (\pi_1(\pi_2 l)), \pi_2(\pi_2 l) \rangle : B \times [A]^n \rightarrow B \times [A]^{n-1}$,
- ...
- $\Delta, g : A \rightarrow B \rightarrow B \vdash_{\mathcal{T}_g} \lambda l. \langle g(\pi_1 l) (\pi_1(\pi_2 l)), \pi_2(\pi_2 l) \rangle : B \times [A]^1 \rightarrow B \times \text{Bool}$

3 Linear λ -calculus with Polymorphic Iteration: System \mathcal{L}_g

In this section we give the syntax, reduction rules and typing rules of a linear version of System \mathcal{T}_g that we call System \mathcal{L}_g . A linear version of System \mathcal{T} with monomorphic iterators is discussed in [6].

3.1 Terms and Linearity

The set of linear λ -terms is built from: variables x, y, \dots ; linear abstraction $\lambda x. t$, where $x \in \text{fv}(t)$; and application tu , where $\text{fv}(t) \cap \text{fv}(u) = \emptyset$. Here $\text{fv}(t)$ denotes the set of free variables of t . Because x is used at least once in the body of the abstraction, and the condition on the application ensures that all variables are used at most once, these terms are syntactically linear (variables occur exactly once in each term).

Since we are in a linear calculus, we cannot have the usual notion of pairs and projections; instead, we have pairs and splitters:

$$\begin{aligned} \langle t, u \rangle & \quad \text{if } \text{fv}(t) \cap \text{fv}(u) = \emptyset \\ \text{let } \langle x, y \rangle = t \text{ in } u & \quad \text{if } x, y \in \text{fv}(u) \text{ and } \text{fv}(t) \cap \text{fv}(u) = \emptyset \end{aligned}$$

Note that when projecting from a pair, we use both projections. A simple example is the swapping function: $\lambda x. \text{let } \langle y, z \rangle = x \text{ in } \langle z, y \rangle$.

Finally, we have booleans `true` and `false`, with a linear conditional: `cond t u v` where $\text{fv}(t) \cap \text{fv}(u) = \emptyset$ and $\text{fv}(u) = \text{fv}(v)$; and numbers (built from 0 and S), with a linear iterator: `iter t u v` where $\text{fv}(t) \cap \text{fv}(u) = \text{fv}(u) \cap \text{fv}(v) = \text{fv}(v) \cap \text{fv}(t) = \emptyset$. $S^n 0$ denotes n applications of S to 0. Table 1 summarises the syntax of System \mathcal{L}_g .

Construction	Variable Constraint	Free Variables (fv)
0, true, false	–	\emptyset
$S t$	–	$\text{fv}(t)$
$\text{iter } t u v$	$\text{fv}(t) \cap \text{fv}(u) = \text{fv}(u) \cap \text{fv}(v) = \emptyset$ $\text{fv}(t) \cap \text{fv}(v) = \emptyset$	$\text{fv}(t) \cup \text{fv}(u) \cup \text{fv}(v)$
x	–	$\{x\}$
$t u$	$\text{fv}(t) \cap \text{fv}(u) = \emptyset$	$\text{fv}(t) \cup \text{fv}(u)$
$\lambda x.t$	$x \in \text{fv}(t)$	$\text{fv}(t) \setminus \{x\}$
$\langle t, u \rangle$	$\text{fv}(t) \cap \text{fv}(u) = \emptyset$	$\text{fv}(t) \cup \text{fv}(u)$
$\text{let } \langle x, y \rangle = t \text{ in } u$	$\text{fv}(t) \cap \text{fv}(u) = \emptyset, x, y \in \text{fv}(u)$	$\text{fv}(t) \cup (\text{fv}(u) \setminus \{x, y\})$
$\text{cond } t u v$	$\text{fv}(u) = \text{fv}(v), \text{fv}(t) \cap \text{fv}(u) = \emptyset$	$\text{fv}(t) \cup \text{fv}(u)$

Table 1 Terms

3.2 Closed Reduction

The dynamics of the system is given by a set of conditional reduction rules (which can be seen as a higher-order membership conditional rewrite system, see [39, 40]). The conditions on the rewrite rules ensure that *Beta* only applies to redexes where the argument is a closed term (which implies that α -conversion is not needed to implement substitution), and only closed functions are iterated. Table 2 gives the reduction rules for System $\mathcal{L}_{\mathcal{G}}$, substitution is a meta-operation defined as usual. Reductions can take place in any context.

Name	Reduction	Condition
<i>Beta</i>	$(\lambda x.t)v \longrightarrow t[v/x]$	$\text{fv}(v) = \emptyset$
<i>Let</i>	$\text{let } \langle x, y \rangle = \langle t, u \rangle \text{ in } v \longrightarrow (v[t/x])[u/y]$	$\text{fv}(t) = \text{fv}(u) = \emptyset$
<i>Cond</i>	$\text{cond true } u v \longrightarrow u$	
<i>Cond</i>	$\text{cond false } u v \longrightarrow v$	
<i>Iter</i>	$\text{iter } (S t) u v \longrightarrow v(\text{iter } t u v)$	$\text{fv}(v) = \emptyset$
<i>Iter</i>	$\text{iter } 0 u v \longrightarrow u$	$\text{fv}(v) = \emptyset$

Table 2 Closed reduction

We give some examples to illustrate the system.

Example 3 – Erasing numbers: although we are in a linear system, we can erase numbers by using them in iterators. We define the projection functions on pairs of numbers as follows:

$$\begin{aligned} \text{fst} &= \lambda x.\text{let } \langle u, v \rangle = x \text{ in iter } v u (\lambda z.z) \\ \text{snd} &= \lambda x.\text{let } \langle u, v \rangle = x \text{ in iter } u v (\lambda z.z) \end{aligned}$$

- Copying numbers. The following function takes a number n and returns a pair $\langle n, n \rangle$:
 $C = \lambda x.\text{iter } x \langle 0, 0 \rangle (\lambda x.\text{let } \langle a, b \rangle = x \text{ in } \langle S a, S b \rangle)$
- Addition: $\text{add} = \lambda mn.\text{iter } m n (\lambda x.S x)$
- Multiplication: $\lambda mn.\text{iter } m 0 (\text{add } n)$
- Predecessor: $\lambda n.\text{fst}(\text{iter } n \langle 0, 0 \rangle (\lambda x.\text{let } \langle t, u \rangle = C(\text{snd } x) \text{ in } \langle t, S u \rangle))$
- Ackermann: $\text{ack}(m, n) = (\text{iter } m (\lambda x.S x) (\lambda gu.\text{iter } (S u) (S 0) g)) n$

Closed reduction is a weak strategy in the sense that it does not allow us to reduce all terms to full normal form: for instance, $(\lambda x.x)z$ is irreducible. However, closed reduction is

adequate for programs: we refer to [6] for a proof that System \mathcal{T} with closed reduction has the same computational power as System \mathcal{T} .

Although linear, System $\mathcal{L}_{\mathcal{G}}$ is not strongly normalising. For instance, the term $\Omega = \Delta\Delta$ where $\Delta = \lambda x.\text{iter } S^2 0 (\lambda xy.xy) (\lambda y.yx)$ is non-terminating: $\Omega \longrightarrow^* \Omega$. We will define a linear type system for System $\mathcal{L}_{\mathcal{G}}$ and show that typable terms are strongly normalisable.

3.3 Type System

To type the terms in System $\mathcal{L}_{\mathcal{G}}$ we use a set of *linear types*. *Iterative types* will serve to type the functions used in iterators.

Definition 2 (Types) The set of *linear types* is defined by the grammar:

$$A, B ::= \text{Nat} \mid \text{Bool} \mid \alpha \mid A \multimap B \mid A \otimes B$$

where Nat and Bool are the types of numbers and booleans, and α ranges over type variables.

Let A_0, \dots, A_n be a (non-empty) list of linear types. $It(A_0, \dots, A_n)$ denotes a non-empty set of *linear iterative types* defined by induction on n , as before, but considering linear types. As before, $It(A_0) = It(A_0, A_0) = It(A_0, \dots, A_0)$.

The typing rules specifying how to assign types to untyped terms are given in Figure 2, where we use the following abbreviations: $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} t : It(A_0, \dots, A_n)$ iff $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} t : B$ for each $B \in It(A_0, \dots, A_n)$. It is a Curry-style type system (there are no type decorations in terms). We do not have Weakening and Contraction rules: we are in a linear system; the logical rules split the context between the premises. For terms of the form $\text{iter } t u v$, we check that t is a term of type Nat and that v and u are compatible.

All the functions given in Example 3 above can be typed in a straightforward way. We give more examples below.

Example 4 For the first example, consider the term

$$D = \lambda z.\text{iter } (S^2 0) (\lambda xy.\langle x, y \rangle) (\lambda x.xz)$$

which allows us to copy arbitrary closed terms in System $\mathcal{L}_{\mathcal{G}}$ (i.e., for any closed term t , $D t \longrightarrow^* \langle t, t \rangle$, see [3] for more details), is typable. We show the type for D , which illustrates the use of iterative types. In the following N denotes Nat and B denotes $A \otimes A$.

$$\frac{\frac{\frac{\vdash_{\mathcal{L}_{\mathcal{G}}} S^2 0 : \text{N}}{\vdash_{\mathcal{L}_{\mathcal{G}}} \lambda xy.\langle x, y \rangle : A \multimap A \multimap B}}{z : A \vdash_{\mathcal{L}_{\mathcal{G}}} (\lambda x.xz) : It(A \multimap A \multimap B, A \multimap B, B)}}{z : A \vdash_{\mathcal{L}_{\mathcal{G}}} \text{iter } (S^2 0) (\lambda xy.\langle x, y \rangle) (\lambda x.xz) : B}}{\vdash_{\mathcal{L}_{\mathcal{G}}} \lambda z.\text{iter } (S^2 0) (\lambda xy.\langle x, y \rangle) (\lambda x.xz) : A \multimap B}$$

Note that $z : A \vdash_{\mathcal{L}_{\mathcal{G}}} (\lambda x.xz) : (A \multimap A \multimap B) \multimap (A \multimap B)$ and $z : A \vdash_{\mathcal{L}_{\mathcal{G}}} (\lambda x.xz) : (A \multimap B) \multimap B$. Therefore $z : A \vdash_{\mathcal{L}_{\mathcal{G}}} (\lambda x.xz) : It(A \multimap A \multimap B, A \multimap B, B)$.

The second example uses the notation defined before for arrays. We can define a replicate function that creates n copies of a closed term u :

$$\text{replicate } u = \underbrace{[u, \dots, u]}_n$$

Axiom and Structural Rule:

$$\frac{}{x : A \vdash_{\mathcal{L}_g} x : A} \text{ (Axiom)} \quad \frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathcal{L}_g} t : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathcal{L}_g} t : C} \text{ (Exchange)}$$

Logical Rules:

$$\frac{\Gamma, x : A \vdash_{\mathcal{L}_g} t : B}{\Gamma \vdash_{\mathcal{L}_g} \lambda x. t : A \multimap B} \text{ (}\multimap\text{Intro)} \quad \frac{\Gamma \vdash_{\mathcal{L}_g} t : A \multimap B \quad \Delta \vdash_{\mathcal{L}_g} u : A}{\Gamma, \Delta \vdash_{\mathcal{L}_g} tu : B} \text{ (}\multimap\text{Elim)}$$

$$\frac{\Gamma \vdash_{\mathcal{L}_g} t : A \quad \Delta \vdash_{\mathcal{L}_g} u : B}{\Gamma, \Delta \vdash_{\mathcal{L}_g} \langle t, u \rangle : A \otimes B} \text{ (}\otimes\text{Intro)} \quad \frac{\Gamma \vdash_{\mathcal{L}_g} t : A \otimes B \quad \Delta, x : A, y : B \vdash_{\mathcal{L}_g} u : C}{\Gamma, \Delta \vdash_{\mathcal{L}_g} \text{let } \langle x, y \rangle = t \text{ in } u : C} \text{ (}\otimes\text{Elim)}$$

Numbers

$$\frac{}{\vdash_{\mathcal{L}_g} 0 : \text{Nat}} \text{ (Zero)} \quad \frac{\Gamma \vdash_{\mathcal{L}_g} n : \text{Nat}}{\Gamma \vdash_{\mathcal{L}_g} S n : \text{Nat}} \text{ (Succ)}$$

$$\frac{\Gamma \vdash_{\mathcal{L}_g} t : \text{Nat} \quad \Theta \vdash_{\mathcal{L}_g} u : A_0 \quad \Delta \vdash_{\mathcal{L}_g} v : It(A_0, \dots, A_n) \quad (\star)}{\Gamma, \Theta, \Delta \vdash_{\mathcal{L}_g} \text{iter } t \ u \ v : A_n} \text{ (Iter)}$$

(\star) where if $t \equiv S^m 0$ then $n = m$ otherwise $n = 0$

Booleans

$$\frac{}{\vdash_{\mathcal{L}_g} \text{true} : \text{Bool}} \text{ (True)} \quad \frac{}{\vdash_{\mathcal{L}_g} \text{false} : \text{Bool}} \text{ (False)}$$

$$\frac{\Delta \vdash_{\mathcal{L}_g} t : \text{Bool} \quad \Gamma \vdash_{\mathcal{L}_g} u : A \quad \Gamma \vdash_{\mathcal{L}_g} v : A}{\Gamma, \Delta \vdash_{\mathcal{L}_g} \text{cond } t \ u \ v : A} \text{ (Cond)}$$

Fig. 2 Type System for System \mathcal{L}_g

We encode replicate in System \mathcal{L}_g as: $\text{replicate} = \lambda u. \text{iter } S^0 \text{ true } \lambda l. \langle u, l \rangle$. The resulting term is typable in System \mathcal{L}_g : $\Gamma, \Delta \vdash_{\mathcal{L}_g} \text{replicate} : A \multimap [A]^n$, if:

- $\Delta, u : A \vdash_{\mathcal{L}_g} \lambda l. \langle u, l \rangle : \text{Bool} \multimap [A]^1$,
- ...
- $\Delta, u : A \vdash_{\mathcal{L}_g} \lambda l. \langle u, l \rangle : [A]^{n-1} \multimap [A]^n$

The duplication term D can also be generalised for any n , although replicate is a more compact term. In [6], a duplication term is defined without the use of polymorphic iteration. The resulting term is much bigger than D or replicate, and depends on the type of the term being copied. For example to duplicate a number we define the term $D^{\text{Nat}} : \text{Nat} \multimap \text{Nat} \otimes \text{Nat}$ as:

$$\lambda x. \text{iter } (S^2 0) \langle \mathcal{M}(\text{Nat}), \mathcal{M}(\text{Nat}) \rangle (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, \text{Nat}) \langle w, x \rangle)$$

where $\mathcal{M}(\text{Nat})$ creates a term of type Nat , and $\mathcal{E}(z, \text{Nat})$, creates a term to erase a term z of type Nat . In this case $\mathcal{M}(\text{Nat}) = 0$ and $\mathcal{E}(z, \text{Nat}) = \text{iter } z (\lambda x. x) (\lambda x. x)$. Note that, this term requires more reduction steps to duplicate the term, since we have to erase the created terms of type Nat , which are used as temporary placeholders for the copies of the term.

We denote by $\text{dom}(\Gamma)$ the set of variables such that $x_i : A_i \in \Gamma$. Since there are no Weakening and Contraction rules, we have:

Lemma 1 *If $\Gamma \vdash_{\mathcal{L}_g} t : A$ then $\text{dom}(\Gamma) = \text{fv}(t)$.*

Proof Induction on the type derivation for $\Gamma \vdash_{\mathcal{L}_g} t : A$. □

Even though the typing rules for iterators in System $\mathcal{L}_{\mathcal{G}}$ are more liberal than those of System \mathcal{T} , the system is still confluent, reductions preserve types, and typable terms are strongly normalisable. We prove these properties in the remainder of this section.

3.4 Reduction preserves types

Lemma 2 (Substitution) *If $\Gamma, x : A \vdash_{\mathcal{L}_{\mathcal{G}}} t : B$ and $\Delta \vdash_{\mathcal{L}_{\mathcal{G}}} u : A$, where $\text{fv}(t) \cap \text{fv}(u) = \emptyset$, then $\Gamma, \Delta \vdash_{\mathcal{L}_{\mathcal{G}}} t[u/x] : B$.*

Proof By induction on the type derivation $\Gamma, x : A \vdash_{\mathcal{L}_{\mathcal{G}}} t : B$. Note that, by Lemma 1, $x \in \text{fv}(t)$. \square

Theorem 1 (Subject Reduction) *If $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} M : A$ and $M \longrightarrow N$, then $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} N : A$.*

Proof By induction on the type derivation $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} M : A$, using Lemma 2. \square

3.5 Strong normalisation

In System $\mathcal{L}_{\mathcal{G}}$, every sequence of reductions starting from a typable term is finite (i.e. typable terms are strongly normalisable). We show the strong normalisation property for System $\mathcal{L}_{\mathcal{G}}$, based on the strong normalisation property for Gödel's System \mathcal{T} [19]. We first define a translation from System $\mathcal{L}_{\mathcal{G}}$ into System \mathcal{T} .

Definition 3 We define the compilation of types and terms in System $\mathcal{L}_{\mathcal{G}}$ into System \mathcal{T} , denoted $J \cdot K$, in the following way:

$$\begin{array}{ll}
J\text{Nat}K & = \text{Nat} & J\text{Bool}K & = \text{Bool} \\
J A \multimap B K & = J A K \multimap J B K & J A \otimes B K & = J A K \otimes J B K \\
J 0 K & = 0 \\
J \text{true} K & = \text{true} \\
J \text{false} K & = \text{false} \\
J S t K & = S(J t K) \\
J x K & = x \\
J \lambda y. t K & = \lambda y. J t K \\
J t u K & = J t K J u K \\
J \langle t, u \rangle K & = \langle J t K, J u K \rangle \\
J \text{let } \langle x, y \rangle = t \text{ in } u K & = J u K [(\pi_1 J t K)/x][(\pi_2 J t K)/y] \\
J \text{cond } t \ u \ v K & = \text{cond } J t K \ J u K \ J v K \\
J \text{iter } t \ u \ v K & = \begin{cases} J v K^m (J u K) & \text{if } t = S^m 0, m > 0 \\ \text{iter } J t K \ J u K \ J v K & \text{otherwise} \end{cases}
\end{array}$$

Also, if $\Gamma = x_1 : A_1, \dots, x_n : A_n$, then $J \Gamma K = x_1 : J A_1 K, \dots, x_n : J A_n K$.

Note that the translation of an iterator where the number of times to iterate is known and positive, develops this iteration. If it is zero or not known we use System \mathcal{T} 's iterator.

Lemma 3 *Let t and u be terms in System $\mathcal{L}_{\mathcal{G}}$ such that $\text{fv}(u) \cap \text{fv}(t) = \emptyset$, then:*

$$J t K [J u K / x] \longrightarrow^* J t [u/x] K$$

Proof By induction on the structure of t . \square

Lemma 4 *If $\Gamma \vdash_{\mathcal{L}_g} t : T$, then $J\Gamma K \vdash_{\mathcal{G}} JtK : JT$.*

Proof By induction on $\Gamma \vdash_{\mathcal{L}_g} t : T$. \square

Definition 4 Consider the set of non-negative integers \mathbb{N} , and the set $\Omega = \mathbb{N} \cup \{\top\}$. Let $>$ be the usual order relation on integers \mathbb{N} . Let $>$ on Ω be such that $\forall x \in \mathbb{N}, \top > x$. Let \gg on $\mathbb{N} \times \Omega$ be defined as:

$$(x_1, y_1) \gg (x_2, y_2) \text{ iff } x_1 > x_2 \vee (x_1 = x_2 \wedge y_1 > y_2)$$

Definition 5 A multiset M over a set A is a function $M : A \rightarrow \mathbb{N}$. Let $M(x)$ be the number of copies of $x \in A$ in M . We use the standard set notation (e.g. $\{a, a, b\}$ for the function $\{a \rightarrow 2, b \rightarrow 1, c \rightarrow 0\}$) and:

- Element:** $x \in M \Leftrightarrow M(x) > 0$
- Inclusion:** $M \subseteq N \Leftrightarrow \forall x \in A. M(x) \leq N(x)$
- Union:** $(M \cup N)(x) = M(x) + N(x)$
- Difference:** $(M - N)(x) = M(x) \dot{-} N(x)$ where $m \dot{-} n = \max(0, m - n)$

Definition 6 For any term t let $\text{depth}(t)$ be defined as:

$$\begin{aligned} \text{depth}(0) &= \text{depth}(\text{true}) = \text{depth}(\text{false}) = \text{depth}(x) = 0 \\ \text{depth}(\lambda y.t) &= \text{depth}(S t) = \text{depth}(t) \\ \text{depth}(tu) &= \text{depth}(\langle t, u \rangle) = \text{depth}(\text{let } \langle x, y \rangle = t \text{ in } u) = \max\{\text{depth}(t), \text{depth}(u)\} \\ \text{depth}(\text{cond } t \ u \ v) &= \max\{\text{depth}(t), \text{depth}(u), \text{depth}(v)\} \\ \text{depth}(\text{iter } n \ u \ v) &= \max\{\text{depth}(n), \text{depth}(u), \text{depth}(v)\} + 1 \end{aligned}$$

Definition 7 For any term t let $|t|$ be a multiset over $\mathbb{N} \times \Omega$ defined as:

$$\begin{aligned} |0| &= |\text{true}| = |\text{false}| = |x| = \emptyset \\ |\lambda y.t| &= |S t| = |t| \\ |tu| &= |\langle t, u \rangle| = |\text{let } \langle x, y \rangle = t \text{ in } u| = |t| \cup |u| \\ |\text{cond } t \ u \ v| &= |t| \cup |u| \cup |v| \\ |\text{iter } n \ u \ v| &= \begin{cases} |u| \cup |v| \cup \{(\text{depth}(\text{iter } n \ u \ v) + 1, m)\} & \text{if } n = S^m 0, m > 0 \\ |n| \cup |u| \cup |v| \cup \{(\text{depth}(\text{iter } n \ u \ v) + 1, \top)\} & \text{otherwise} \end{cases} \end{aligned}$$

Given the set of terms Λ , let $\mathcal{M}(\Lambda) = \{|t| \mid t \in \Lambda\}$.

Let $>_{\text{mul}}$ be the usual multiset extension to $\mathcal{M}(\Lambda)$ of the relation \gg defined above. Since \gg is well founded, so is $>_{\text{mul}}$ (see [7]).

Definition 8 We use $\longrightarrow_{\text{iter}}$ to represent the reduction

$$\text{iter } S^{m+1} 0 \ u \ v \longrightarrow_{\text{iter}} v(\text{iter } S^m 0 \ u \ v).$$

Lemma 5 *If $t \longrightarrow_{\text{iter}} t'$ then $\text{depth}(t) \geq \text{depth}(t')$ and $|t| >_{\text{mul}} |t'|$. Therefore $\longrightarrow_{\text{iter}}$ is terminating.*

Proof By induction on the structure of t . We only show a few cases for $t = \text{iter } n \ u \ v$.

– $\text{iter } S^{m+1}(0) u v \longrightarrow_{\text{Iter}} v(\text{iter } S^m(0) u v)$.

$$\begin{aligned} \text{depth}(\text{iter } S^{m+1}(0) u v) &= \max\{\text{depth}(u), \text{depth}(v)\} + 1 \\ &= \max\{\text{depth}(v), \text{depth}(u), \text{depth}(v)\} + 1 \\ &= \text{depth}(v(\text{iter } S^m(0) u v)) \end{aligned}$$

Let $b = \max\{\text{depth}(u), \text{depth}(v)\}$, and notice that for any $(x, y) \in |v|$, $x < b + 1$.

$$\begin{aligned} |\text{iter } S^{m+1}(0) u v| &= |u| \cup |v| \cup \{(b+1, m+1)\} \\ |v(\text{iter } S^m(0) u v)| &= |v| \cup |u| \cup |v| \cup \{(b+1, m)\} \end{aligned}$$

Consider $X = \{(b+1, m+1)\}$, $Y = |v| \cup \{(b+1, m)\}$

$$\begin{aligned} (|\text{iter } S^{m+1}(0) u v| - X) \cup Y &= ((|u| \cup |v| \cup \{(b+1, m+1)\}) - \\ &\quad \{(b+1, m+1)\}) \cup (|v| \cup \{(b+1, m)\}) \\ &= |u| \cup |v| \cup |v| \cup \{(b+1, m)\} \\ &= |v(\text{iter } S^m(0) u v)| \end{aligned}$$

– $\text{iter } n u v \longrightarrow_{\text{Iter}} \text{iter } n' u v$, because $n \longrightarrow_{\text{Iter}} n'$. By induction $\text{depth}(n) \geq \text{depth}(n')$.

$$\begin{aligned} \text{depth}(\text{iter } n u v) &= \max\{\text{depth}(n), \text{depth}(u), \text{depth}(v)\} \\ &\geq \max\{\text{depth}(n'), \text{depth}(u), \text{depth}(v)\} \\ &= \text{depth}(\text{iter } n' u v) \end{aligned}$$

Let $b = \max\{\text{depth}(n), \text{depth}(u), \text{depth}(v)\}$. Notice that $b \geq b'$, and $b' = \max\{\text{depth}(n'), \text{depth}(u), \text{depth}(v)\}$. By induction $|n| >_{\text{mul}} |n'|$, thus $\exists X, Y$. $|n'| = (|n| - X) \cup Y$.

$$\begin{aligned} |\text{iter } n u v| &= |n| \cup |u| \cup |v| \cup \{(b+1, \top)\} \\ |\text{iter } n' u v| &= |n'| \cup |u| \cup |v| \cup \{(b'+1, \top)\} \end{aligned}$$

Let $X' = X \cup \{(b+1, \top)\}$ and $Y' = Y \cup \{(b'+1, \top)\}$ if $b > b'$, and $X' = X$ and $Y' = Y$ if $b = b'$. In either case:

$$(|\text{iter } n u v| - X') \cup Y' = |\text{iter } n' u v|.$$

The other cases for iter are proved in a similar way. \square

Corollary 1 Every sequence of $\longrightarrow_{\text{Iter}}$ starting from t is finite.

Proof By contradiction. Suppose there is an infinite sequence

$$t_1 \longrightarrow_{\text{Iter}} t_2 \longrightarrow_{\text{Iter}} \cdots$$

By Lemma 5 there is an infinite descending chain $|t_1| >_{\text{mul}} |t_2| >_{\text{mul}} \cdots$, which is absurd since $>_{\text{mul}}$ is well-founded. \square

Lemma 6 1. If $t \longrightarrow_{\text{Iter}} t'$, then $\text{JtK} = \text{Jt}'\text{K}$

2. If $t \longrightarrow t'$ (other than $\longrightarrow_{\text{Iter}}$), then $\text{JtK} \longrightarrow^+ \text{Jt}'\text{K}$

Proof The first part is obtained from the diagram:

$$\begin{array}{ccc} \text{iter } S^{m+1} 0 u v & \xrightarrow{\text{Iter}} & v(\text{iter } S^m 0 u v) \\ \text{J}\cdot\text{K} \downarrow & & \downarrow \text{J}\cdot\text{K} \\ \text{JvK}^{m+1}(\text{JuK}) & = & \text{JvK}(\text{JvK}^m(\text{JuK})) \end{array}$$

The second part is obtained by inspection of the different reduction rules. We show only the *Beta* and *Iter* rules.

β -redexes:

$$\begin{array}{ccc} (\lambda x.t)u & \xrightarrow{\quad} & t[u/x] \\ \text{J}\cdot\text{K} \downarrow & & \downarrow \text{J}\cdot\text{K} \\ (\lambda x.\text{JtK})\text{JuK} & \rightarrow \text{JtK}[\text{JuK}/x] \xrightarrow[\text{(Lem 3)}]{*} & \text{Jt}[u/x]\text{K} \end{array}$$

Iterator:

$$\begin{array}{ccc} \text{iter } 0 u v & \xrightarrow{\quad} & u & \quad & \text{iter } S t u v & \xrightarrow{\quad} & v(\text{iter } t u v) \\ \text{J}\cdot\text{K} \downarrow & & \downarrow \text{J}\cdot\text{K} & \text{and} & \downarrow \text{J}\cdot\text{K} & & \downarrow \text{J}\cdot\text{K} \\ \text{iter } 0 \text{JuK JvK} & \rightarrow & \text{JuK} & & \text{iter } S(\text{JtK}) \text{JuK JvK} & \rightarrow & \text{JvK}(\text{iter } \text{JtK JuK JvK}) \end{array}$$

□

Lemma 7 *If JtK is strongly normalisable, so is t .*

Proof By contradiction. Since $\xrightarrow{\text{Iter}}$ is terminating, an infinite derivation out of t must contain an infinite number of \rightarrow . By Lemma 6 we obtain an infinite derivation for JtK which gives a contradiction. □

Theorem 2 (Strong Normalisation) *If $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} t : T$, t is strongly normalisable.*

Proof Consequence of Lemma 4, SN of System \mathcal{T} and Lemma 7. □

3.6 Church-Rosser

System $\mathcal{L}_{\mathcal{G}}$ is confluent, which implies that normal forms are unique. For typable terms, confluence is a direct consequence of strong normalisation and the fact that the rules are non-overlapping (using Newman's Lemma [35]). In fact, all System $\mathcal{L}_{\mathcal{G}}$ terms are confluent even if they are non-terminating: that can be proved using the Tait-Martin-Löf method. We refer to [2] for a detailed proof.

Theorem 3 \rightarrow is Church-Rosser.

Corollary 2 (Uniqueness of Normal Form) *A term t in System $\mathcal{L}_{\mathcal{G}}$ has at most one normal form.*

4 Linear Type Reconstruction

Recall that there are non-terminating linear terms, for instance: $\Omega = \Delta\Delta$ where

$$\Delta = \lambda x.\text{iter } S^2 0 (\lambda xy.xy) (\lambda y.yx)$$

is non-terminating, since there is a reduction sequence $\Omega \longrightarrow^* \Omega$.

Ω is not typable in System $\mathcal{L}_{\mathcal{G}}$. In this section we develop a type reconstruction algorithm for System $\mathcal{L}_{\mathcal{G}}$ that can be used to rule out non-terminating terms such as the one above. The algorithm is in a similar style to that of Damas-Milner [13]. We begin by giving an alternative presentation of the type assignment rules for System $\mathcal{L}_{\mathcal{G}}$, which will suggest a type reconstruction algorithm. We will prove it to be both sound and complete with respect to these rules. We refer the reader to [34, 13, 12] for background to this work.

4.1 A Hybrid Type System

System $\mathcal{L}_{\mathcal{G}}$ is a resource sensitive calculus, and we place a restriction on the use of assumptions in a derivation: namely use them all exactly once. Its type system is given in a multiplicative style, where each term is provided with the exact number of type assumptions for its free variables (see Figure 2). Following [30, 31, 22], in this section we will simulate a multiplicative system using a hybrid (between multiplicative and additive) presentation of the rules. We will write typing judgements in the following way:

$$\Gamma \mid \Theta \vdash_{\mathcal{L}}^H t : A$$

where Γ and Θ are lists such that the elements in Θ are also in Γ , and $\Gamma \setminus \Theta$ contains precisely the assumptions necessary to type t ; we call Γ the *before-set* and Θ the *after-set*, indicating that the derivation uses the assumptions only in $\Gamma \setminus \Theta$. The idea is best explained by an example. Consider the rule:

$$\frac{\Gamma \mid \Delta \vdash_{\mathcal{L}}^H t : A \multimap B \quad \Delta \mid \Theta \vdash_{\mathcal{L}}^H u : A}{\Gamma \mid \Theta \vdash_{\mathcal{L}}^H tu : B} (\multimap\text{Elim})$$

This rule states that if we type tu using Γ then Θ will be left over. We give t all of the assumptions, and the remaining Δ are given to u . The ones that are not consumed here are exactly those which are left over in typing tu . The rationale for choosing this notation will become more apparent when we present the type reconstruction algorithm. Note that a similar technique for dealing with multiplicative contexts was used in [21], in the context of proof search for intuitionistic logic, for splitting contexts lazily when attempting to prove a tensor.

The full type assignment for System $\mathcal{L}_{\mathcal{G}}$ using the “before-and-after” presentation is given in Figure 3, where we write $\Gamma, x : A$ to denote the list obtained by adding to Γ the element $x : A$ at the end (and in general we write Γ, Δ for list concatenation), and $x : A \in \Gamma$ holds if $x : A$ is the last assumption for x in the list Γ . The notation $\Gamma \setminus \{x : A\}$ represents the list Γ where we have deleted the last assumption for x (and in general, $\Gamma \setminus \Delta$ denotes the list Γ without the elements in Δ).

To relate the two versions of the type system (Figures 2 and 3) we need some lemmas, where we use the following notation: if $\Gamma \mid \Delta$ is a type environment in the hybrid system, then we write $\bar{\Gamma} \mid \bar{\Delta}$ to denote any permutation of Γ that preserves the relative order of assumptions for the same variable (that is, all the assumptions for x occur in the same order in Γ and $\bar{\Gamma}$) and the corresponding sub-list $\bar{\Delta}$.

Axiom:

$$\frac{x : A \in \Gamma}{\Gamma \mid \Gamma \setminus \{x : A\} \vdash_{\mathcal{L}}^H x : A} \text{ (Axiom)}$$

Logical Rules:

$$\frac{\Gamma, x : A \mid \Delta \vdash_{\mathcal{L}}^H t : B}{\Gamma \mid \Delta \vdash_{\mathcal{L}}^H \lambda x.t : A \multimap B} \text{ (}\multimap\text{Intro)} \quad \frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H t : A \multimap B \quad \Gamma' \mid \Delta \vdash_{\mathcal{L}}^H u : A}{\Gamma \mid \Delta \vdash_{\mathcal{L}}^H tu : B} \text{ (}\multimap\text{Elim)}$$

$$\frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H t : A \quad \Gamma' \mid \Delta \vdash_{\mathcal{L}}^H u : B}{\Gamma \mid \Delta \vdash_{\mathcal{L}}^H (t, u) : A \otimes B} \text{ (}\otimes\text{Intro)}$$

$$\frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H t : A \otimes B \quad \Gamma', x : A, y : B \mid \Delta \vdash_{\mathcal{L}}^H u : C}{\Gamma \mid \Delta \vdash_{\mathcal{L}}^H \text{let } (x, y) = t \text{ in } u : C} \text{ (}\otimes\text{Elim)}$$

Numbers

$$\frac{}{\Gamma \mid \Gamma \vdash_{\mathcal{L}}^H 0 : \text{Nat}} \text{ (Zero)} \quad \frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H n : \text{Nat}}{\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H S n : \text{Nat}} \text{ (Succ)}$$

$$\frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H t : \text{Nat} \quad \Gamma' \mid \Theta \vdash_{\mathcal{L}}^H u : A_0 \quad \Theta \mid \Delta \vdash_{\mathcal{L}}^H v : It(A_0, \dots, A_n) \quad (\star)}{\Gamma \mid \Delta \vdash_{\mathcal{L}}^H \text{iter } t u v : A_n} \text{ (Iter)}$$

(\star) where if $t \equiv S^m 0$ then $n = m$ otherwise $n = 0$

Booleans

$$\frac{}{\Gamma \mid \Gamma \vdash_{\mathcal{L}}^H \text{true} : \text{Bool}} \text{ (True)} \quad \frac{}{\Gamma \mid \Gamma \vdash_{\mathcal{L}}^H \text{false} : \text{Bool}} \text{ (False)}$$

$$\frac{\Gamma \mid \Delta \vdash_{\mathcal{L}}^H t : \text{Bool} \quad \Delta \mid \Theta \vdash_{\mathcal{L}}^H u : A \quad \Delta \mid \Theta \vdash_{\mathcal{L}}^H v : A}{\Gamma \mid \Theta \vdash_{\mathcal{L}}^H \text{cond } t u v : A} \text{ (Cond)}$$

Fig. 3 Hybrid Type System for System $\mathcal{L}_{\mathcal{G}}$

Lemma 8 (Permutations) *If $\Gamma \mid \Delta \vdash_{\mathcal{L}}^H t : A$, then $\overline{\Gamma} \mid \overline{\Delta} \vdash_{\mathcal{L}}^H t : A$.*

Proof By induction on the derivation. In the permutation, only the relative order of the assumptions for x is relevant in the Axiom. \square

Lemma 9 (Monotonicity) *$\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H t : A$ if and only if $\Delta, \Gamma \mid \Delta, \Gamma' \vdash_{\mathcal{L}}^H t : A$.*

Proof By induction on the type derivation. \square

As a consequence of these lemmas (since the elements in Γ' are also in Γ):

$$\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H t : A \text{ implies } \Gamma \setminus \Gamma' \mid \emptyset \vdash_{\mathcal{L}}^H t : A$$

The relationship between the multiplicative and the hybrid versions of System $\mathcal{L}_{\mathcal{G}}$ is as follows:

Theorem 4 – *If $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} t : A$ then $\overline{\Gamma} \mid \emptyset \vdash_{\mathcal{L}}^H t : A$ for any permutation $\overline{\Gamma}$.*
– *If $\overline{\Gamma} \mid \emptyset \vdash_{\mathcal{L}}^H t : A$ for some permutation $\overline{\Gamma}$ then $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} t : A$.*

Proof (\Rightarrow) By induction on the type derivation, using the previous lemmas. We distinguish cases according to the last rule applied; some interesting cases are:

- Exchange: Since the type environment contains the same elements in the premise and conclusion, the result follows directly by induction.
- \multimap Intro: By induction, $\bar{\Gamma}, x : A \mid \emptyset \vdash_{\mathcal{L}}^H t : B$, for any permutation of $\Gamma, x : A$. In particular, $\bar{\Gamma}, x : A \mid \emptyset \vdash_{\mathcal{L}}^H t : B$, and the result follows using \multimap Intro in the hybrid system.
- lter: By induction, $\bar{\Gamma} \mid \emptyset \vdash_{\mathcal{L}}^H t : \text{Nat}$, $\bar{\Theta} \mid \emptyset \vdash_{\mathcal{L}}^H u : A_0$, and $\bar{\Delta} \mid \emptyset \vdash_{\mathcal{L}}^H v : \text{It}(A_0, \dots, A_n)$. By Monotonicity, $\bar{\Delta}, \bar{\Theta}, \bar{\Gamma} \mid \bar{\Delta}, \bar{\Theta} \vdash_{\mathcal{L}}^H t : \text{Nat}$ and $\bar{\Delta}, \bar{\Theta} \mid \bar{\Theta} \vdash_{\mathcal{L}}^H u : A_0$. Then, using rule lter in the hybrid system we obtain: $\bar{\Delta}, \bar{\Theta}, \bar{\Gamma} \mid \emptyset \vdash_{\mathcal{L}}^H \text{iter } t u v : \text{It}(A_0, \dots, A_n)$. The result follows using the Permutation lemma.

(\Leftarrow) We assume $\bar{\Gamma} \mid \emptyset \vdash_{\mathcal{L}}^H t : A$ for some permutation, and proceed by induction on t , using the previous lemmas. Again, we distinguish cases according to the last rule applied, and show only some interesting cases.

- \multimap Elim: The premises are: $\bar{\Gamma} \mid \Gamma' \vdash_{\mathcal{L}}^H t : A \multimap B$, and $\Gamma' \mid \emptyset \vdash_{\mathcal{L}}^H u : A$, then by Monotonicity we also have $\bar{\Gamma} \setminus \Gamma' \mid \emptyset \vdash_{\mathcal{L}}^H t : A \multimap B$. By induction: $\Gamma \setminus \Gamma' \vdash_{\mathcal{L}} t : A$ and $\Gamma' \vdash_{\mathcal{L}} u : A$, then $\Gamma, \Gamma' \vdash_{\mathcal{L}} t u : B$, using \multimap Elim in the multiplicative version.
- lter: The premises are: $\bar{\Gamma} \mid \Gamma' \vdash_{\mathcal{L}}^H t : \text{Nat}$, and $\Gamma' \mid \Theta \vdash_{\mathcal{L}}^H u : A_0$, and $\Theta \mid \emptyset \vdash_{\mathcal{L}}^H v : \text{It}(A_0, \dots, A_n)$. Then by Monotonicity we also have $\bar{\Gamma} \setminus \Gamma' \mid \emptyset \vdash_{\mathcal{L}}^H t : \text{Nat}$, and $\Gamma' \setminus \Theta \mid \emptyset \vdash_{\mathcal{L}}^H u : A_0$. By induction: $\Gamma \setminus \Gamma' \vdash_{\mathcal{L}} t : \text{Nat}$, $\Gamma' \setminus \Theta \vdash_{\mathcal{L}} u : A_0$, and $\Theta \vdash_{\mathcal{L}} v : \text{It}(A_0, \dots, A_n)$. Since $\Gamma = \Gamma \setminus \Gamma' \cup \Gamma' \setminus \Theta \cup \Theta$, the result follows using lter in the multiplicative version, and the Permutation lemma. \square

4.2 The Type Reconstruction Algorithm \mathcal{L}

Our presentation of the algorithm \mathcal{L} will assume that the terms are syntactically linear. It is a trivial extension to the algorithm to perform this kind of checking—we just need extra conditions to be satisfied.

We will need unification of types in this section, a simple extension to the unification algorithm used in Damas-Milner’s system, based on a variant of Robinson’s theorem [37]. The definition is standard (see for instance [33]). Substitutions are mappings from type variables to types. They are idempotent; composition is denoted by juxtaposition. We assume that $\text{mgu}(A, B)$ gives the *most general unifier* of A and B , that is, a substitution U such that:

- $UA = UB$;
- if V also unifies A and B then V is a substitution instance of U , i.e. $V = SU$ for some substitution S ; and
- U only involves variables in A and B —no new variables are introduced during unification.

If A, B are not unifiable then $\text{mgu}(A, B)$ fails.

Our type reconstruction algorithm will take as input a term and a list of type assumptions for variables. To reflect the linearity constraint that all assumptions must be used exactly once, we treat type assumptions as resources—once an assumption is used, we remove it. To this end our type reconstruction algorithm will return a triple (rather than a pair as in the case of \mathcal{W}), which consists of a substitution, a type, and the assumptions not yet used.

We write R, S to range over substitutions, α, β to range over type variables, Γ, Γ' to range over lists of assumptions. We write id for the identity substitution, and substitution over lists is defined element-wise. For a substitution R , we write $R(\Gamma \mid \Gamma')$ for $R\Gamma \mid R\Gamma'$, and define substitution on judgements by: $R(\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H t : A) = R\Gamma \mid R\Gamma' \vdash_{\mathcal{L}}^H t : RA$. We assume that the function *new* returns a fresh type variable each time it is called.

Definition 9 (Type Reconstruction Algorithm \mathcal{L})

$\mathcal{L}(\Gamma, e) = (T, \tau, \Gamma')$ where:

1. If e is the identifier x , and $x : A \in \Gamma$ then $T = \text{id}$, $\tau = A$, $\Gamma' = \Gamma \setminus \{x : A\}$.
2. If e is of the form $\langle t, u \rangle$, let

$$\begin{aligned} (R, A, \Gamma_1) &= \mathcal{L}(\Gamma, t) \\ (S, B, \Gamma_2) &= \mathcal{L}(R\Gamma_1, u) \end{aligned}$$

then $T = SR$, $\tau = SA \otimes B$, $\Gamma' = \Gamma_2$.

3. If e is of the form $\text{let } \langle x, y \rangle = t \text{ in } u$, let

$$\begin{aligned} (R, A, \Gamma_1) &= \mathcal{L}(\Gamma, t) \\ U &= \text{mgu}(A, \alpha \otimes \beta); \quad \alpha, \beta \text{ new} \\ (S, B, \Gamma_2) &= \mathcal{L}((UR\Gamma_1, x : U\alpha, y : U\beta), u) \end{aligned}$$

then $T = SUR$, $\tau = B$, $\Gamma' = \Gamma_2$.

4. If e is of the form $\lambda x.t$, let

$$(R, B, \Gamma_1) = \mathcal{L}((\Gamma, x : \alpha), t); \quad \alpha \text{ new}$$

then $T = R$, $\tau = R\alpha \multimap B$, $\Gamma' = \Gamma_1$.

5. If e is of the form tu , let

$$\begin{aligned} (R, C, \Gamma_1) &= \mathcal{L}(\Gamma, t) \\ (S, A, \Gamma_2) &= \mathcal{L}(R\Gamma_1, u) \\ U &= \text{mgu}(SC, A \multimap \beta); \quad \beta \text{ new} \end{aligned}$$

then $T = USR$, $\tau = U\beta$, $\Gamma' = \Gamma_2$.

6. If e is 0 then $T = \text{id}$, $\tau = \text{Nat}$ and $\Gamma' = \Gamma$.
7. If e is $S t$, and $\mathcal{L}(\Gamma, t) = (R, A, \Gamma_1)$, and $\text{mgu}(A, \text{Nat}) = U$, then $T = UR$, $\tau = \text{Nat}$ and $\Gamma' = \Gamma_1$.
8. If e is of the form $\text{iter } t u v$, where $t \neq S^m 0$ for $m > 0$, let

$$\begin{aligned} (R, C, \Gamma_1) &= \mathcal{L}(\Gamma, t) \\ U &= \text{mgu}(C, \text{Nat}) \\ (S, A, \Gamma_2) &= \mathcal{L}(UR\Gamma_1, u) \\ (T', B, \Gamma_3) &= \mathcal{L}(S\Gamma_2, v) \\ V &= \text{mgu}(B, T'(A \multimap A)) \end{aligned}$$

then $T = VT'SUR$, $\tau = VT'A$, $\Gamma' = \Gamma_3$.

9. If e is of the form $\text{iter } (S^m 0) u v$ where $m > 0$, let

$$\begin{aligned} (S, B, \Gamma_0) &= \mathcal{L}(\Gamma, u) \\ (R, A, \Gamma_1) &= \mathcal{L}(S\Gamma_0, v) \\ B_0 &= RB \\ S_0 &= RS \\ \text{for } i = 1 \dots m \{ & U_i = \text{mgu}(A, B_{i-1} \multimap \beta_i); \quad \beta_i \text{ new} \\ & B_i = U_i \beta_i \} \\ \text{for } i = 1 \dots m \{ & U_m^i = U_m \dots U_i \\ & S_i = U_m^i S_0 \} \end{aligned}$$

$$\text{Condition : } S_1 \Gamma_0 = \dots = S_m \Gamma_0 \text{ and } S_1 \Gamma_1 = \dots = S_m \Gamma_1$$

then $T = S_1$, $\tau = B_m$, $\Gamma' = \Gamma_1$.

10. If e is true or false then $T = \text{id}$, $\tau = \text{Bool}$ and $\Gamma' = \Gamma$.
 11. If e is of the form $\text{cond } t \ u \ v$, let

$$\begin{aligned} (R, C, \Gamma_1) &= \mathcal{L}(\Gamma, t) \\ U &= \text{mgu}(C, \text{Bool}) \\ (S, A, \Gamma_2) &= \mathcal{L}(UR\Gamma_1, u) \\ (S', B, \Gamma_3) &= \mathcal{L}(SUR\Gamma_1, v) \\ \text{Condition} &: \Gamma_3 = S\Gamma_2 \\ V &= \text{mgu}(B, S'A) \end{aligned}$$

then $T = VS'SUR$, $\tau = VB$, $\Gamma' = \Gamma_3 (= S\Gamma_2)$.

Note that \mathcal{L} fails if it is not one of the above forms.

Cases 1-7, 10 and 11 are standard for a linear λ -calculus with numbers, booleans and pairs (see [31]). Cases 8 and 9 deal with iterator terms. In case 8 we first check that t can be given type Nat , then type u with the remaining assumptions, and finally type v using only the assumptions not consumed in the typing of t and u , checking that v has an arrow type of the correct form. The interesting case is 9: here we deal with an iterator term in which the number of iterations is known. We type u and v as in case 8, and then check that v can be given a set of iterative types.

Soundness of \mathcal{L} . If the algorithm \mathcal{L} succeeds in typing a term e under some assumptions, then we want to be sure that e actually is typable. This is called Soundness and states that our algorithm is safe—it produces no wrong results.

Lemma 10 (Substitution) *If there is a derivation $\Gamma \mid \Gamma' \vdash_{\mathcal{L}}^H e : \tau$ then, for any substitution S , there is also a derivation for $S(\Gamma \mid \Gamma') \vdash_{\mathcal{L}}^H e : S\tau$.*

Proof By induction over the length of the derivation. □

Theorem 5 (Soundness of \mathcal{L}) *If $\mathcal{L}(\Gamma, e)$ succeeds with (S, τ, Γ') then there is a derivation of $S(\Gamma \mid \Gamma') \vdash_{\mathcal{L}}^H e : \tau$.*

Proof By induction on the structure of terms e , using the Substitution Lemma and the fact that substitutions are idempotent. We show the most interesting cases:

1. If e is of the form $\langle t, u \rangle$ then $\mathcal{L}(\Gamma, t)$ succeeds with (R, A, Γ_1) and $\mathcal{L}(R\Gamma_1, u)$ succeeds with (S, B, Γ_2) . By induction twice, there are derivations $R(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}}^H t : A$ and $S(R\Gamma_1 \mid \Gamma_2) \vdash_{\mathcal{L}}^H u : B$. Since Γ_2 is included in $R\Gamma_1$, and R is idempotent, also $SR(\Gamma_1 \mid \Gamma_2) \vdash_{\mathcal{L}}^H u : B$. By Lemma 10 we can write the first derivation as $SR(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}}^H t : SA$. Now, by \otimes Intro $SR(\Gamma \mid \Gamma_2) \vdash_{\mathcal{L}}^H e : SA \otimes B$.
2. If e is of the form $\text{iter } t \ u \ v$ and $t \neq S^m 0$ for $m > 0$, then $\mathcal{L}(\Gamma, t)$ succeeds with (R, C, Γ_1) and $\text{mgu}(C, \text{Nat})$ succeeds with a substitution U . $\mathcal{L}(UR\Gamma_1, u)$ succeeds with (S, A, Γ_2) , $\mathcal{L}(S\Gamma_2, v)$ succeeds with (T', B, Γ_3) , and $\text{mgu}(B, T'A \multimap T'A)$ succeeds with a substitution V . Now by induction and Lemma 10 there are derivations ending in $VT'SUR(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}}^H t : \text{Nat}$, $VT'SUR(\Gamma_1 \mid \Gamma_2) \vdash_{\mathcal{L}}^H u : VT'A$, and $VT'SUR(\Gamma_2 \mid \Gamma_3) \vdash_{\mathcal{L}}^H v : VT'A \multimap VT'A$, and the result follows by the Iter rule.
3. If e is of the form $\text{iter } (S^m 0) \ u \ v$ where $m > 0$, then $\mathcal{L}(\Gamma, u)$ succeeds with (S, B, Γ_0) , and $\mathcal{L}(S\Gamma_0, v)$ succeeds with (R, A, Γ_1) , where $B_0 = RB$ and $S_0 = RS$. By induction and Lemma 10 there are derivations ending in $S_0\Gamma \mid S_0\Gamma_0 \vdash_{\mathcal{L}}^H u : B_0$, and $R(S\Gamma_0 \mid \Gamma_1) \vdash_{\mathcal{L}}^H v : A$. Since Γ_1 is included in $S\Gamma_0$, and S is idempotent, also $RS(\Gamma_0 \mid \Gamma_1) \vdash_{\mathcal{L}}^H v : A$. Now,

for $i = 1 \dots m$, and using Lemma 10 there are derivations ending in $S_i \Gamma_0 \mid S_i \Gamma_1 \vdash_{\mathcal{L}}^H v : U_m^i A = U_m^i B_{i-1} \multimap U_m^{i+1} B_i$. Since $S_1 \Gamma_0 = \dots = S_m \Gamma_0$, and $S_1 \Gamma_1 = \dots = S_m \Gamma_1$, then $S_1 \Gamma_0 \mid S_1 \Gamma_1 \vdash_{\mathcal{L}}^H v : It(U_m^1 B_0, U_m^2 B_1, \dots, U_m^m B_{m-1}, B_m)$, and $S_1 \Gamma \mid S_1 \Gamma_0 \vdash_{\mathcal{L}}^H u : U_m^1 B_0$. Therefore, since $S_1 \Gamma \mid S_1 \Gamma \vdash_{\mathcal{L}}^H S^m 0 : \text{Nat}$, then $S_1(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}}^H e : B_m$, follows by the Iter rule.

4. If e is of the form $\text{cond } t \ u \ v$, then $\mathcal{L}(\Gamma, t)$ succeeds with (R, C, Γ_1) and $\text{mgu}(C, \text{Bool})$ succeeds with a substitution U . $\mathcal{L}(UR\Gamma_1, u)$ succeeds with (S, A, Γ_2) , $\mathcal{L}(SUR\Gamma_1, v)$ succeeds with (S', B, Γ_3) , and $\text{mgu}(B, S'A)$ succeeds with a substitution V . Now by induction, Lemma 10, and because $S\Gamma_2 = \Gamma_3$, there are derivations ending in $VS'SUR(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}}^H t : \text{Bool}$, $VS'SUR(\Gamma_1 \mid \Gamma_3) \vdash_{\mathcal{L}}^H u : VB$, and $VS'SUR(\Gamma_1 \mid \Gamma_3) \vdash_{\mathcal{L}}^H v : VB$ and the result follows by the Cond rule. \square

Corollary 3 *If $\mathcal{L}(\Gamma, e)$ succeeds with (S, τ, Γ') then there is a derivation*

$$S(\Gamma) \setminus S(\Gamma') \vdash_{\mathcal{L}} e : \tau$$

Proof Consequence of Theorem 5, and Lemmas 8 and 9. \square

Completeness of \mathcal{L} . If a term can be typed using the inference rules, then we would require our algorithm to also be able to compute the type of this term. The proof follows closely the proof of the completeness of \mathcal{W} in [12]. Note that a notion of *principal* type follows as an immediate corollary of the completeness theorem.

Theorem 6 (Completeness of \mathcal{L}) *If there is a derivation $S(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}}^H e : \tau$ for some substitution S , then:*

1. $\mathcal{L}(\Gamma, e)$ succeeds with $(R, A, S\Gamma_1)$ for some R, A .
2. There exists a substitution T such that: $TR(\Gamma \mid \Gamma_1) = S(\Gamma \mid \Gamma_1)$ and $TA = \tau$.

Proof By induction over the structure of e . We will only show case (9) (iteration), where $e = \text{iter}(S^m 0) \ u \ v$ with $m > 0$. For the other cases the proof is similar to standard proofs of completeness of type assignment algorithms.

In case (9), we have $\Gamma \mid \Gamma \vdash_{\mathcal{L}}^H S^m 0 : \text{Nat}$, and we also have, by the induction hypothesis,

$$\mathcal{L}(\Gamma, u) = (R_u, A_u, \Theta)$$

for some R_u, A_u , and there exists a substitution T_u such that $T_u R_u(\Gamma \mid \Theta) = S(\Gamma \mid \Theta)$ and $T_u A_u = A_0$. Now, note that $S\Theta$ is also an instance of $R_u \Theta$, thus it is in the conditions of the theorem, and thus by the induction hypothesis $\mathcal{L}(R_u \Theta, v)$ succeeds with (R_v, A, Γ_1) for some R_v, A .

We will now use induction on m , and consider first the base case ($m = 1$). By the induction hypothesis, there is a substitution T_v such that $T_v A = A_0 \multimap A_1$ and $T_v R_v R_u(\Theta \mid \Delta) = S(\Theta \mid \Delta)$. We now show that, if β_0 is a new type variable, then $\text{mgu}(A, R_v A_u \multimap \beta_0)$ succeeds. We will do this by showing that

$$U_0 = \{A_1 / \beta_0\} \cup T_v$$

is a unifying substitution. Now

$$U_0 A = (\{A_1 / \beta_0\} \cup T_v) A = T_v A = A_0 \multimap A_1$$

To see that we also have $U_0(R_v A_u \multimap \beta_0) = A_0 \multimap A_1$ we first note that since

$$S\Theta = T_v R_v R_u \Theta = T_u R_u \Theta$$

we have $T_v R_v \alpha = T_u \alpha$ for any type variable α occurring free in $R_u \Theta$. But then, since every type variable occurring free in $R_v A_u$ occurs free in $R_v R_u \Theta$, we have

$$U_0(R_v A_0 \multimap \beta_0) = (U_0 R_v A_u) \multimap A_1$$

but since β_0 is new

$$U_0 R_v A_u = T_v R_v A_u = T_u A_u = A_0$$

Since U_0 unifies A and $R_v A_u \multimap \beta_0$, $\text{mgu}(A, R_v A_u \multimap \beta_0)$ succeeds with substitution U , and thus $\mathcal{L}(\Gamma, e)$ also succeeds with $(UR_v R_u, U\beta_0, \Gamma_1)$ as we wanted.

To prove the second clause of the theorem let W be a substitution such that

$$U_0 = WU$$

Then

$$\begin{aligned} WUR_v R_u(\Gamma \mid \Delta) &= U_0 R_v R_u(\Gamma \mid \Delta) \\ &= T_v R_v R_u(\Gamma \mid \Delta) \\ &= S(\Gamma \mid \Delta) \end{aligned}$$

Note also that $U_0 \beta_0 = WU \beta_0 = A_1$.

Now, the induction step ($m > 1$) follows by using a similar reasoning to the one used in the base case: We show that if there is a type derivation $\Theta \mid \Delta \vdash_{\mathcal{L}_{\mathcal{G}}} v : It(A_0, \dots, A_m)$ for $m > 1$, then there is a substitution S such that for $i = 0 \dots m$, $SU_m^{i+1} B_i = A_i$, where

$$\begin{aligned} (S', B, \Gamma_0) &= \mathcal{L}(\Gamma, u) \\ (R, A, \Gamma_1) &= \mathcal{L}(S' \Gamma_0, v) \\ B_0 &= RB \\ S_0 &= RS' \\ \text{for } i = 1 \dots m \{ &U_i = \text{mgu}(A, B_{i-1} \multimap \beta_i); \beta_i \text{ new} \\ &B_i = U_i \beta_i \} \\ \text{for } i = 1 \dots m \{ &U_m^i = U_m \dots U_i \\ &S_i = U_m^i S_0 \} \end{aligned}$$

$$\text{and } S_1 \Gamma_0 = \dots = S_m \Gamma_0 \text{ and } S_1 \Gamma_1 = \dots = S_m \Gamma_1 \square$$

Some complexity considerations over the type reconstruction algorithm follow: in System $\mathcal{L}_{\mathcal{G}}$, the copy operator D has type $A \multimap A \otimes A$. If we apply D to an expression M with type B , then the application DM will be typed by $B \otimes B$. Thus applying D to a typable expression doubles the length of its type.

Now, by iterating the copy combinator D to any typable expression M ($\text{iter } n M D$), we can prove that for arbitrary large n , $(\text{iter } n M D)$ has type with length $2^{\Omega(n)}$. Thus the type inference algorithm, although it makes a number of calls to unification that is polynomial in the size of the input, has parameters passed to these calls which are exponentially bounded, thus it is exponential in the size of the input.

5 Iterative Types

In this section we present two intersection type systems closely related to System $\mathcal{L}_{\mathcal{G}}$. The first one is based on Damas' type system [12], a less known polymorphic type system with the same power of the Hindley-Milner system. The second is based on rank-2 intersection types [8, 24].

Iterative types are a compact way of expressing several type derivations for an iterated function. Consider the iterator function itself $\lambda x. \text{iter } t \ u \ x$. When this function is applied to a term v our type rules assume that v must be typed with every type in $It(A_0, \dots, A_n)$. Another way to see it is to type $\text{iter } t \ u \ x$ with multiple assumptions for x , and then type v with all the elements of the set of types declared for x . One standard way to extend a type system by allowing multiple assumptions for free variables is by using intersection types.

5.1 Polymorphic iteration

In the calculus presented here, which we call System \mathcal{L}_\cap^1 , intersection types are only used in the set of assumptions for free variables. This kind of restriction to intersection type systems was first used in Damas's PhD thesis [12] in the definition of a system (later called Damas's System T [20]) with the same set of typable expressions as the widely known Hindley-Milner system, but that instead of using \forall -quantified types, allows multiple types in the set of assumptions for each free variable. We will use a similar method to type iterators.

We consider the set *Types*, of linear types defined in Section 3. Let S range over the set of all finite non-empty subsets of *Types*. The set *Iter* of *intersection types* is defined as follows: $\bar{A} ::= \wedge S$.

The type environments (or bases in the terminology of intersection systems) of the systems presented in this section represent a total function from the set of variables of the term to *Iter*. Bases that associate to term-variables elements of *Types* will be called *monomorphic*.

System \mathcal{L}_\cap^1 is obtained from System $\mathcal{L}_\mathcal{G}$ by replacing the rule for (Iter) by the two rules given in Figure 4.

$$\frac{\Gamma \vdash_{\mathcal{L}_\cap^1} t : \text{Nat} \quad \Delta \vdash_{\mathcal{L}_\cap^1} u : A_0}{\Gamma, x : \wedge It(A_0, \dots, A_n), \Delta \vdash_{\mathcal{L}_\cap^1} \text{iter } t \ u \ x : A_n} \text{ (VarIter)}$$

(\star) where if $t \equiv S^m 0$ then $n = m$ otherwise $n = 0$

$$\frac{\Gamma, x : \wedge S \vdash_{\mathcal{L}_\cap^1} \text{iter } t \ u \ x : A \quad \forall B_i \in S. \Delta \vdash_{\mathcal{L}_\cap^1} v : B_i}{\Gamma, \Delta \vdash_{\mathcal{L}_\cap^1} \text{iter } t \ u \ v : A} \text{ (Iter)}$$

Fig. 4 System $\mathcal{L}_\mathcal{G}$ with Intersection Types

System \mathcal{L}_\cap^1 allows multiple types in the set of assumptions for each free variable. This can be seen as using intersection types for free variables and System \mathcal{L}_\cap^1 can be seen as a system of rank-1 intersection types.

We now show two results relating System $\mathcal{L}_\mathcal{G}$ and System \mathcal{L}_\cap^1 .

Theorem 7 *If there is a derivation $\Gamma \vdash_{\mathcal{L}_\mathcal{G}} e : \tau$, then $\Gamma \vdash_{\mathcal{L}_\cap^1} e : \tau$*

Proof We show the case for Iter, as the other cases are trivial by induction.

If e is of the form $\text{iter } t \ u \ v$, then $\Gamma, \Theta, \Delta \vdash_{\mathcal{L}_\mathcal{G}} \text{iter } t \ u \ v : A_n$ if $\Gamma \vdash_{\mathcal{L}_\mathcal{G}} t : \text{Nat}$, $\Theta \vdash_{\mathcal{L}_\mathcal{G}} u : A_0$ and $\Delta \vdash_{\mathcal{L}_\mathcal{G}} v : It(A_0, \dots, A_n)$, where if $t \equiv S^m 0$ then $n = m$ otherwise $n = 0$. By induction, $\Gamma \vdash_{\mathcal{L}_\cap^1} t : \text{Nat}$ and $\Theta \vdash_{\mathcal{L}_\cap^1} u : A_0$. Therefore by VarIter and Exchange, $\Gamma, \Theta, x : \wedge It(A_0, \dots, A_n) \vdash_{\mathcal{L}_\cap^1} \text{iter } t \ u \ x : A_n$. Again by induction, $\forall B_i \in It(A_0, \dots, A_n). \Delta \vdash_{\mathcal{L}_\cap^1} v : B_i$. Thus, by Iter $\Gamma, \Theta, \Delta \vdash_{\mathcal{L}_\cap^1} \text{iter } t \ u \ v : A_n$. \square

This last result shows that any term typable in System $\mathcal{L}_{\mathcal{G}}$ is also typable in \mathcal{L}_{\cap}^1 . The opposite does not hold, i.e. system \mathcal{L}_{\cap}^1 allows more typings than System $\mathcal{L}_{\mathcal{G}}$. In particular, when typing an open term of the form $\text{iter}(S^m 0) u x$, it allows x to have an iterative type $It(A_0, \dots, A_n)$. For example, we can have the following derivation in System \mathcal{L}_{\cap}^1 (consider for example, $\Gamma = \{x : \wedge It(\text{Nat} \multimap \text{Nat} \multimap \text{Nat} \otimes \text{Nat}, \dots, \text{Nat} \otimes \text{Nat})\}$):

$$\Gamma \vdash_{\mathcal{L}_{\cap}^1} (\lambda y. \text{fst } y)(\text{iter}(S^2 0) (\lambda x_1 x_2. \langle x_1, x_2 \rangle) x) : \text{Nat}$$

but the term $(\lambda y. \text{fst } y)(\text{iter}(S^2 0) (\lambda x_1 x_2. \langle x_1, x_2 \rangle) x)$ is not typable in System $\mathcal{L}_{\mathcal{G}}$, because, for $(\text{iter}(S^2 0) (\lambda x_1 x_2. \langle x_1, x_2 \rangle) x)$, we can only have derivations of the form (consider $\Gamma = \{x : (A \multimap A \multimap A \otimes A) \multimap (A \multimap A \multimap A \otimes A)\}$):

$$\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} \text{iter}(S^2 0) (\lambda x_1 x_2. \langle x_1, x_2 \rangle) x : A \multimap A \multimap A \otimes A.$$

Note however that, if the bases used in derivations in System \mathcal{L}_{\cap}^1 are monomorphic, then those terms are also typable in System $\mathcal{L}_{\mathcal{G}}$.

Theorem 8 *If there is a derivation $\Gamma \vdash_{\mathcal{L}_{\cap}^1} e : \tau$, with Γ monomorphic, then $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} e : \tau$.*

Proof We only show the case for Iter , as the other cases are trivial by induction.

If e is of the form $\text{iter } t u v$, then $\Gamma, \Delta \vdash_{\mathcal{L}_{\cap}^1} \text{iter } t u v : A_n$ if

$$\Gamma, x : \wedge It(A_0, \dots, A_n) \vdash_{\mathcal{L}_{\cap}^1} \text{iter } t u x : A_n \quad \Delta \vdash_{\mathcal{L}_{\cap}^1} v : It(A_0, \dots, A_n)$$

Also, $\Gamma, x : \wedge It(A_0, \dots, A_n) \vdash_{\mathcal{L}_{\cap}^1} \text{iter } t u x : A_n$ if $\Gamma' \vdash_{\mathcal{L}_{\cap}^1} t : \text{Nat}$ and $\Gamma'' \vdash_{\mathcal{L}_{\cap}^1} u : A_0$ where if $t \equiv S^m 0$ then $n = m$ otherwise $n = 0$, and $\Gamma = \Gamma', \Gamma''$. By induction hypothesis: $\Gamma' \vdash_{\mathcal{L}_{\mathcal{G}}} t : \text{Nat}$ $\Gamma'' \vdash_{\mathcal{L}_{\mathcal{G}}} u : A_0$ $\Delta \vdash_{\mathcal{L}_{\mathcal{G}}} v : It(A_0, \dots, A_n)$. Thus, by Iter $\Gamma, \Delta \vdash_{\mathcal{L}_{\mathcal{G}}} \text{iter } t u v : A_n$. \square

In particular for closed terms, the two systems are equivalent.

Corollary 4 $\vdash_{\mathcal{L}_{\mathcal{G}}} e : \tau$ iff $\vdash_{\mathcal{L}_{\cap}^1} e : \tau$.

Therefore, we have a type inference algorithm for \mathcal{L}_{\cap}^1 closed terms.

5.2 Rank 2 Intersection Types: System \mathcal{L}_{\cap}^2

To fully exploit the power of intersection types when typing terms of the form $\text{iter}(S^m 0) u x$, we now extend the type system \mathcal{L}_{\cap}^1 in order to abstract on variables typed with intersection types.

The rank 2 intersection type assignment for System $\mathcal{L}_{\mathcal{G}}$ (which we call System \mathcal{L}_{\cap}^2) is obtained from System \mathcal{L}_{\cap}^1 , by replacing the rules $\multimap\text{Intro}$ and $\multimap\text{Elim}$ by the two rules given in Figure 5. Note that we do not distinguish the types $\wedge\{A\}$ and A . This system corresponds to a linear version of a rank 2 intersection type system with iterators, and it includes System $\mathcal{L}_{\mathcal{G}}$ (and System \mathcal{L}_{\cap}^1).

Theorem 9 *If there is a derivation $\Gamma \vdash_{\mathcal{L}_{\mathcal{G}}} e : \tau$, then $\Gamma \vdash_{\mathcal{L}_{\cap}^2} e : \tau$*

Proof Trivial. Note that, $\mathcal{L}_{\cap}^1 \subset \mathcal{L}_{\cap}^2$, since the rules $\multimap\text{Intro}$ and $\multimap\text{Elim}$ of System \mathcal{L}_{\cap}^1 , are a subcase of the same rules in System \mathcal{L}_{\cap}^2 . \square

$$\frac{\Gamma, x : \wedge S \vdash_{\mathcal{L}_\cap^2} t : B}{\Gamma \vdash_{\mathcal{L}_\cap^2} \lambda x. t : \wedge S \multimap B} \text{ (}\multimap\text{Intro)}$$

$$\frac{\Gamma \vdash_{\mathcal{L}_\cap^2} t : \wedge S \multimap B \quad \forall A_i \in S. \Delta \vdash_{\mathcal{L}_\cap^2} u : A_i}{\Gamma, \Delta \vdash_{\mathcal{L}_\cap^2} tu : B} \text{ (}\multimap\text{Elim)}$$

Fig. 5 Rank 2 Intersection Types version of System $\mathcal{L}_\mathcal{G}$

System \mathcal{L}_\cap^2 is stronger than System \mathcal{L}_\cap^1 (therefore, than System $\mathcal{L}_\mathcal{G}$), since it allows abstractions on polymorphic variables. Note however, that polymorphic variables are only introduced through VarIter. For example, we can have the following typing in System \mathcal{L}_\cap^2 :

$$\vdash_{\mathcal{L}_\cap^2} (\lambda y. (\lambda x. \text{fst } x) (\text{iter } (S^2 0) (\lambda x_1 x_2. x_1 x_2) y)) (\lambda z. z (S^3 0)) : \text{Nat}$$

but this term is not typable in System $\mathcal{L}_\mathcal{G}$. Note also that System \mathcal{L}_\cap^2 allows us to write more compact versions of admissible linear terms. Consider for example F to be a closed function with types $It(\text{Nat} \multimap \text{Nat} \multimap \text{Nat} \otimes \text{Nat}, \dots, \text{Nat} \otimes \text{Nat})$, then

$$(\lambda f p. \text{cond } p (\text{iter } (S^2 0) 0 f) (\text{iter } (S^2 0) (S 0) f)) F$$

is typable in System \mathcal{L}_\cap^2 .

Subject reduction for Systems \mathcal{L}_\cap^1 and \mathcal{L}_\cap^2 is proved in a similar way as for System $\mathcal{L}_\mathcal{G}$. Confluence for untyped terms was proved in [3], and this result, together with subject reduction, implies confluence for terms typable in Systems \mathcal{L}_\cap^1 and \mathcal{L}_\cap^2 .

Summarising, we have shown how iterative types are related with intersection types, which in turn shows the expressiveness of System $\mathcal{L}_\mathcal{G}$. The relation between the set of terms typable in the three systems is: $\mathcal{L}_\mathcal{G} \subset \mathcal{L}_\cap^1 \subset \mathcal{L}_\cap^2$. As shown in Corollary 4, iterative types correspond to a restriction of linear rank-2 intersection types where intersections are allowed only in the typing context and all terms are closed (therefore programs).

6 Conclusions

We have studied the use of polymorphic iteration, focusing on a linear version of Gödel's System \mathcal{T} with a new type construct. We have shown that polymorphic iterators are related to intersection types, and form a decidable subsystem. We have given a type reconstruction algorithm. Since we proved that the calculus is strongly normalising, type reconstruction has the usual applications. The results relating iterative types and intersection types, together with the results in [3,6] which show that System $\mathcal{L}_\mathcal{G}$ can simulate Gödel's System \mathcal{T} , indicate that System $\mathcal{L}_\mathcal{G}$ actually corresponds to a version of System \mathcal{T} with a restricted form of intersection types.

Acknowledgements We would like to thank the anonymous referees for insightful comments that have helped to improve this paper. This work was partially funded by LIACC through Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia (FCT).

References

1. S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. S. Alves. *Linearisation of the Lambda Calculus*. PhD thesis, Faculty of Science - University of Porto, April 2007. Available from <http://www.dcc.fc.up.pt/~sandra/papers/PhDthesis.pdf.gz>.
3. S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of linear functions. In Z. Ésik, editor, *Proceedings of the 15th EACSL Conference on Computer Science Logic (CSL'06)*, volume 4207 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2006.
4. S. Alves, M. Fernández, M. Florido, and I. Mackie. Iterator types. In H. Seidl, editor, *Proceedings FOSSACS*, volume 4423 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 2007.
5. S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of closed-reduction strategies. *Electr. Notes Theor. Comput. Sci.*, 174(10):57–74, 2007. Proceedings of WRS 2006, 6th International Workshop on Rewriting Strategies, FLOC 2006, Seattle.
6. S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel's System T revisited. *Theoretical Computer Science*, 411(11-13):1484–1500, 2010.
7. F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.
8. S. Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Department of Computer Science, University of Nijmegen, 1993.
9. S. Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 151(2):385–435, 1995.
10. H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type-assignment. *J. Symbolic Logic*, 48:931–940, 1983.
11. M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ -calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.
12. L. M. M. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.
13. L. M. M. Damas and R. Milner. Principal type schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.
14. F. Damiani. Rank-2 Intersection and Polymorphic Recursion. In *TLCA'05*, LNCS 3461, pages 146–161. Springer, 2005.
15. M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: explicit substitutions without alpha conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005.
16. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
17. J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. North Holland Publishing Company, Amsterdam, 1989.
18. J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic: Proc. of the Joint Summer Research Conference*, pages 69–108. American Mathematical Society, Providence, RI, 1989.
19. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
20. C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.
21. J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. In *LICS*, pages 32–42. IEEE Computer Society, 1991.
22. J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.*, 110(2):327–365, 1994.
23. P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to Haskell 98. <http://www.haskell.org/tutorial/>, 1999.
24. T. Jim. Rank 2 type systems and recursive definitions. Technical report, Massachusetts Institute of Technology, 1995.
25. T. Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*. ACM Press, 1996.
26. A. J. Kfoury, H. G. Mairson, F. A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proceedings of the 1999 International Conference on Functional Programming*, pages 90–101. ACM Press, 1999.
27. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.

-
28. J. Lambek. From lambda calculus to cartesian closed categories. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–402. Academic Press, London, 1980.
 29. J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics Vol. 7. Cambridge University Press, 1986.
 30. I. Mackie. Lilac: A functional programming language based on linear logic. Master's thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1991.
 31. I. Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.
 32. I. Mackie, L. Román, and S. Abramsky. An internal language for autonomous categories. *Journal of Applied Categorical Structures*, 1(3):311–343, 1993.
 33. A. Martelli and U. Montanari. An efficient unification algorithm. *Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
 34. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
 35. M. Newman. On theories with a combinatorial definition of “equivalence”. *Annals of Mathematics*, 43(2):223–243, 1942.
 36. G. Pottinger. A type assignment for strongly normalizable λ -terms. In *To H.B. Curry, Essays in Combinatory Logic, Lambda-Calculus and Formalism*, pages 535–560. Academic Press, 1980.
 37. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
 38. H. Schwichtenberg. Complexity of normalization in the pure typed lambda-calculus. In *Proceedings L. E. J. Brouwer Centenary Symposium, 1981*, volume 110 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1982.
 39. Y. Toyama. Confluent term rewriting systems with membership. In *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems, CTRS'87, Orsay, France*, volume 308 of *LNCs*, pages 228–241. Springer-Verlag, 1988.
 40. J. Yamada. Confluence of terminating membership conditional trs. In *Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems, CTRS'92, Pont--Mousson, France*, volume 656 of *LNCs*, pages 378–392. Springer-Verlag, 1993.