Iterator Types*

Sandra Alves[1]**, Maribel Fernández[2], Mário Florido[1], and Ian Mackie[2,3]***

[1] University of Porto, Department of Computer Science & LIACC,
R. do Campo Alegre 823, 4150-180, Porto, Portugal
[2] King's College London, Department of Computer Science,
Strand, London, WC2R 2LS, U.K.
[3] LIX, École Polytechnique, 91128 Palaiseau Cedex, France

Abstract. System $\mathcal{L}$ 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 $\mathcal{L}$ 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 $\mathcal{L}$.

1 Introduction

Recently new insights into linearity have lead to the development of rich computational models (see for instance [12, 13, 1, 19, 3]). To support them, new strategies of reduction and new notions of types and typing rules have been introduced.

System $\mathcal{L}$, as defined in [3], extends the linear λ-calculus with numbers, booleans, pairs, and an iterator. Unlike previous linear versions of System $\mathcal{T}$, System $\mathcal{L}$ permits to build an iterator term with an open function, but uses a reduction strategy that will block such subterms until 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 [14], and was used to devise efficient evaluation strategies in the λ-calculus (see [11, 13]).

Although linear systems are known to be computationally weak, System $\mathcal{L}$ has all the power of Gödel's System $\mathcal{T}$ (see [3] for details of the encoding of System $\mathcal{T}$ in System $\mathcal{L}$). 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 one of the keys to the power of System $\mathcal{L}$: in [2] two linear versions of

* Research partially supported by the British Council Treaty of Windsor Grant: "Linearity: Programming Languages and Implementations", and by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *FEDER/POSI*.
** Programa Gulbenkian de Estímulo à Investigação.
*** Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

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.

The other distinctive feature of System $\mathcal{L}$ is the use of a more liberal rule to type iterators, introducing *iterative types*. More precisely, in System $\mathcal{L}$ 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 [18]).

In this paper we study these new types, give a Curry-style type system for System $\mathcal{L}$, and relate it to intersection type assignment systems. Intersection types were introduced by Coppo and Dezani in [7], and since then they have been used to characterise classes of terms with specific normalisation properties (see e.g. [23, 5]), to define type systems with principal typings [17], to define models for the λ-calculus [6], etc. General intersection type assignment systems are undecidable, but several decidable restrictions have been defined (see for example [4, 16, 10]). Iterative types can be seen as a new decidable restriction of intersection types based on iteration. The type system of System $\mathcal{L}$ is decidable: one of the main contributions of this paper is a type reconstruction algorithm for System $\mathcal{L}$.

The rest of this paper is structured as follows. In the next section we recall System $\mathcal{L}$. Section 3 gives a type reconstruction algorithm, including the iterator types, with soundness and completeness proofs. Section 4 contains a detailed analysis of iterator types. Section 5 concludes the paper.

2 Linear λ-calculus with Iterator: System $\mathcal{L}$

In this section we recall the syntax, reduction rules and typing rules of System $\mathcal{L}$ (for more details we refer the reader to [3]).

The set of linear λ-terms is built from: variables $x, y, \ldots$; linear abstraction $\lambda x.t$, where $x \in \mathsf{fv}(t)$; and application tu, where $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$. Here $\mathsf{fv}(t)$ 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:

$$\begin{array}{ll} \langle t, u\rangle & \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \\ \mathtt{let}\ \langle x, y\rangle = t\ \mathtt{in}\ u & \text{if } x, y \in \mathsf{fv}(u) \text{ and } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \end{array}$$

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

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

Construction	**Variable Constraint**	**Free Variables** (fv)
$0, \mathsf{true}, \mathsf{false}$	$-$	$\varnothing$
$\mathsf{S}\ t$	$-$	$\mathsf{fv}(t)$
$\mathsf{iter}\ t\ u\ v$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(v) = \varnothing$ $\mathsf{fv}(t) \cap \mathsf{fv}(v) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(v)$
x	$-$	$\{x\}$
tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) \setminus \{x\}$
$\langle t, u\rangle$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\mathtt{let}\ \langle x, y\rangle = t\ \mathtt{in}\ u$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing,\ x, y \in \mathsf{fv}(u)$	$\mathsf{fv}(t) \cup (\mathsf{fv}(u) \setminus \{x, y\})$
$\mathtt{cond}\ t\ u\ v$	$\mathsf{fv}(u) = \mathsf{fv}(v), \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$

Table 1. Terms

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 [25, 26]). 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 $\mathcal{L}$, substitution is a meta-operation defined as usual. Reductions can take place in any context.

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

Table 2. Closed reduction

We give some examples to illustrate the system.

- Erasing numbers: although we are in a linear system, we can erase numbers by using them in iterators.

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

- Copying numbers: $C = \lambda x.\mathsf{iter}\ x\ \langle 0, 0\rangle\ (\lambda x.\mathtt{let}\ \langle a, b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle)$ takes a number n and returns a pair $\langle n, n\rangle$.
- Addition: $\mathsf{add} = \lambda mn.\mathsf{iter}\ m\ n\ (\lambda x.\mathsf{S}\ x)$

– Multiplication: $\lambda mn.\mathsf{iter}\ m\ 0\ (\mathsf{add}\ n)$
– Predecessor: $\lambda n.\mathsf{fst}(\mathsf{iter}\ n\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle t,u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t,\mathsf{S}\ u\rangle))$
– Ackermann: $ack(m,n) = (\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda gu.\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g))\ n$

To type the terms in System $\mathcal{L}$ we use a set of *linear types*:

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

where Nat and $\mathtt{Bool}$ are the types of numbers and booleans, and α is a type variable.

Let $A_0, \ldots, A_n$ be a (non-empty) list of linear types. $It(A_0, \ldots, A_n)$ denotes a non-empty set of *iterative types* defined by induction on n:

$$\begin{array}{l} n = 0 : It(A_0) = \{A_0 \multimap A_0\} \\ n = 1 : It(A_0, A_1) = \{A_0 \multimap A_1\} \\ n \geq 2 : It(A_0, \ldots, A_n) = It(A_0, \ldots, A_{n-1}) \cup \{A_{n-1} \multimap A_n\} \end{array}$$

Iterative types will serve to type the functions used in iterators. Note that $It(A_0) = It(A_0, A_0) = It(A_0, \ldots, A_0)$.

The typing rules specifying how to assign types to untyped terms are given in Figure 1, where we use the following abbreviations: $\Gamma \vdash_{\mathcal{L}} t : It(A_0, \ldots, A_n)$ iff $\Gamma \vdash_{\mathcal{L}} t : B$ for each $B \in It(A_0, \ldots, A_n)$. 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 $\mathsf{iter}\ t\ u\ v$, we check that t is a term of type Nat and that v and u are compatible. There are two cases: if t is $\mathsf{S}^n 0$ 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 $v : A \multimap A$, for some type A).

All the examples above can be typed in a straightforward way. More interestingly, the term $D = \lambda z.\mathsf{iter}\ (\mathsf{S}^2 0)\ (\lambda xy.\langle x,y\rangle)\ (\lambda x.xz)$ which allows us to copy arbitrary closed terms in System $\mathcal{L}$ (for any closed term t, $D\ t \longrightarrow^* \langle t,t\rangle$, see [3] for more details), is typable. We show a type derivation for D, which illustrates the use of iterative types. In the following type derivation N denotes Nat and B denotes $A \otimes A$.

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

Note that

$$\dfrac{\dfrac{}{x : A \multimap A \multimap B \vdash_{\mathcal{L}} x : A \multimap A \multimap B} \quad \dfrac{}{z : A \vdash_{\mathcal{L}} z : A}}{x : A \multimap A \multimap B, z : A \vdash_{\mathcal{L}} xz : A \multimap B}}{z : A \vdash_{\mathcal{L}} (\lambda x.xz) : (A \multimap A \multimap B) \multimap (A \multimap B)}$$

Axiom and **Structural Rule**:

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

Logical Rules:

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

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

Numbers

$$\frac{}{\vdash_{\mathcal{L}} 0 : \mathsf{Nat}}\ (\mathsf{Zero}) \qquad \frac{\Gamma \vdash_{\mathcal{L}} n : \mathsf{Nat}}{\Gamma \vdash_{\mathcal{L}} \mathsf{S}\ n : \mathsf{Nat}}\ (\mathsf{Succ})$$

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

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

Booleans

$$\frac{}{\vdash_{\mathcal{L}} \mathsf{true} : \mathtt{Bool}}\ (\mathsf{True}) \qquad \frac{}{\vdash_{\mathcal{L}} \mathsf{false} : \mathtt{Bool}}\ (\mathsf{False})$$

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

Fig. 1. Type System for System $\mathcal{L}$

and

$$\dfrac{\dfrac{\overline{x : A \multimap B \vdash_{\mathcal{L}} x : A \multimap B} \qquad \overline{z : A \vdash_{\mathcal{L}} z : A}}{x : A \multimap B, z : A \vdash_{\mathcal{L}} xz : B}}{z : A \vdash_{\mathcal{L}} (\lambda x.xz) : (A \multimap B) \multimap B}$$

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

We recall from [3] that System $\mathcal{L}$ is confluent, reductions preserve types, and typable terms are strongly normalisable[4].

3 Linear Type Reconstruction

This section develops a type reconstruction algorithm for System $\mathcal{L}$. Our algorithm is in a similar style to that of Damas-Milner [9]. We begin by giving a presentation of the type assignment rules 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 [22, 9, 8] for background to this work.

[4] In [3] there are no type variables, but the same results hold here since we don't have instantiation rules.

System $\mathcal{L}$ 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. Following [20], we will simulate a multiplicative system using a hybrid (in between multiplicative and additive) presentation of the rules. We will write typing judgements in the following way:

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

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:

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

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.

The full type assignment for System $\mathcal{L}$ using the "before-and-after" presentation is given in Figure 2, 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 (see Figures 1 and 2) 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 $\overline{\Gamma} \mid \overline{\Delta}$ 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 $\overline{\Gamma}$) and the corresponding sub-list $\overline{\Delta}$.

Lemma 1 (Permutations). *If $\Gamma \mid \Delta \vdash_{\mathcal{L}} t: A$, then $\overline{\Gamma} \mid \overline{\Delta} \vdash_{\mathcal{L}} t: A$.*

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

Lemma 2 (Monotonicity). *$\Gamma \mid \Gamma' \vdash_{\mathcal{L}} t: A$ if and only if $\Delta, \Gamma \mid \Delta, \Gamma' \vdash_{\mathcal{L}} t: A$.*

Proof. By induction on the type derivation.

As a consequence of these lemmas, $\Gamma \mid \Gamma' \vdash_{\mathcal{L}} t : A$ implies $\Gamma \setminus \Gamma' \mid \varnothing \vdash_{\mathcal{L}} t : A$ (since the elements in Γ' are also in Γ).

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

Axiom:

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

Logical Rules:

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

$$\frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}} t : A \qquad \Gamma' \mid \Delta \vdash_{\mathcal{L}} u : B}{\Gamma \mid \Delta \vdash_{\mathcal{L}} \langle t, u \rangle : A \otimes B}\ (\otimes\mathsf{Intro})$$

$$\frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}} t : A \otimes B \qquad \Gamma', x : A, y : B \mid \Delta \vdash_{\mathcal{L}} u : C}{\Gamma \mid \Delta \vdash_{\mathcal{L}} \mathtt{let}\ \langle x, y \rangle = t\ \mathtt{in}\ u : C}\ (\otimes\mathsf{Elim})$$

Numbers

$$\frac{}{\Gamma \mid \Gamma \vdash_{\mathcal{L}} 0 : \mathsf{Nat}}\ (\mathsf{Zero}) \qquad \frac{\Gamma \mid \Gamma' \vdash_{\mathcal{L}} n : \mathsf{Nat}}{\Gamma \mid \Gamma' \vdash_{\mathcal{L}} \mathsf{S}\ n : \mathsf{Nat}}\ (\mathsf{Succ})$$

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

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

Booleans

$$\frac{}{\Gamma \mid \Gamma \vdash_{\mathcal{L}} \mathsf{true} : \mathtt{Bool}}\ (\mathsf{True}) \qquad \frac{}{\Gamma \mid \Gamma \vdash_{\mathcal{L}} \mathsf{false} : \mathtt{Bool}}\ (\mathsf{False})$$

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

Fig. 2. Hybrid Type System for System $\mathcal{L}$

Theorem 1. – *If* $\Gamma \vdash_{\mathcal{L}} t : A$ *then* $\overline{\Gamma} \mid \varnothing \vdash_{\mathcal{L}} t : A$ *for any permutation* $\overline{\Gamma}$.
– *If* $\overline{\Gamma} \mid \varnothing \vdash_{\mathcal{L}} t : A$ *for some permutation* $\overline{\Gamma}$ *then* $\Gamma \vdash_{\mathcal{L}} t : A$.

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, $\overline{\Gamma, x : A} \mid \varnothing \vdash_{\mathcal{L}} t : B$, for any permutation of $\Gamma, x : A$. In particular, $\overline{\Gamma}, x : A \mid \varnothing \vdash_{\mathcal{L}} t : B$, and the result follows using $\multimap$Intro in the hybrid system.
- Iter: By induction, $\overline{\Gamma} \mid \varnothing \vdash_{\mathcal{L}} t : \mathsf{Nat}$, $\overline{\Theta} \mid \varnothing \vdash_{\mathcal{L}} u : A_0$, and $\overline{\Delta} \mid \varnothing \vdash_{\mathcal{L}} v : It(A_0, \ldots, A_n)$. By Monotonicity, $\overline{\Delta}, \overline{\Theta}, \overline{\Gamma} \mid \overline{\Delta}, \overline{\Theta} \vdash_{\mathcal{L}} t : \mathsf{Nat}$ and $\overline{\Delta}, \overline{\Theta} \mid \overline{\Theta} \vdash_{\mathcal{L}}$

$u : A_0$. Then, using rule Iter in the hybrid system we obtain: $\overline{\Delta}, \overline{\Theta}, \overline{\Gamma} \mid \varnothing \vdash_{\mathcal{L}} \mathsf{iter}\ t\ u\ v : It(A_0, \ldots, A_n)$. The result follows using the Permutation lemma.

$\Leftarrow$) We assume $\overline{\Gamma} \mid \varnothing \vdash_{\mathcal{L}} t : A$ 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.

- $\multimap\mathsf{Elim}$: The premises are: $\overline{\Gamma} \mid \Gamma' \vdash_{\mathcal{L}} t : A \multimap B$, and $\Gamma' \mid \varnothing \vdash_{\mathcal{L}} u : A$, then by Monotonicity we also have $\overline{\Gamma} \setminus \Gamma' \mid \varnothing \vdash_{\mathcal{L}} t : A \multimap B$. By induction: $\Gamma \setminus \Gamma' \vdash_{\mathcal{L}} t : A$ and $\Gamma' \vdash_{\mathcal{L}} u : A$, then $\Gamma, \Gamma' \vdash_{\mathcal{L}} tu : B$, using $\multimap\mathsf{Elim}$ in the multiplicative version.
- Iter: The premises are: $\overline{\Gamma} \mid \Gamma' \vdash_{\mathcal{L}} t : \mathsf{Nat}$, and $\Gamma' \mid \Theta \vdash_{\mathcal{L}} u : A_0$, and $\Theta \mid \varnothing \vdash_{\mathcal{L}} v : It(A_0, \ldots, A_n)$. Then by Monotonicity we also have $\overline{\Gamma} \setminus \Gamma' \mid \varnothing \vdash_{\mathcal{L}} t : \mathsf{Nat}$, and $\Gamma' \setminus \Theta \mid \varnothing \vdash_{\mathcal{L}} u : A_0$. By induction: $\Gamma \setminus \Gamma' \vdash_{\mathcal{L}} t : \mathsf{Nat}$, $\Gamma' \setminus \Theta \vdash_{\mathcal{L}} u : A_0$, and $\Theta \vdash_{\mathcal{L}} v : It(A_0, \ldots, A_n)$. Since $\Gamma = \Gamma \setminus \Gamma' \cup \Gamma' \setminus \Theta \cup \Theta$, the result follows using Iter in the multiplicative version, and the Permutation lemma.

3.1 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 [24]. The definition is standard (see for instance [21]). Substitutions are mappings from type variables to types. They are associative and idempotent; composition is denoted by juxtaposition. We assume that $\mathsf{mgu}AB$ 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 the final requirement is that U only involves variables in A and B—no new variables are introduced during unification. If A, B are not unifiable then $\mathsf{mgu}AB$ 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 : $R(\Gamma \mid \Gamma' \vdash_{\mathcal{L}} t : A) = R\Gamma \mid R\Gamma' \vdash_{\mathcal{L}} t : RA$. We assume that the function *new* returns a fresh type variable each time it is called.

Definition 1 (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* $T = \mathsf{id}$, $\tau = A$, $\Gamma' = \Gamma \setminus \{x : A\}$.
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* $\mathtt{let}\ \langle x, y\rangle = t\ \mathtt{in}\ u$, *let*

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

   then $T = SUR, \tau = B, \Gamma' = \Gamma_2$.
4. *If e is of the form $\lambda x.t$, let*

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

   then $T = R, \tau = R\alpha \multimap B, \Gamma' = \Gamma_1$.
5. *If e is of the form tu, let*

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

   then $T = USR, \tau = U\beta, \Gamma' = \Gamma_2$.
6. *If e is* 0 *then* $T = \mathsf{id}$, $\tau = \mathsf{Nat}$ *and* $\Gamma' = \Gamma$.
7. *If e is* $\mathsf{S}\ t$, *and* $\mathcal{L}(\Gamma, t) = (R, A, \Gamma_1)$, *and* $\mathsf{mgu}\ A\ \mathsf{Nat} = U$, *then* $T = UR$, $\tau = \mathsf{Nat}$ *and* $\Gamma' = \Gamma_1$.
8. *If e is of the form* $\mathsf{iter}\ t\ u\ v$, *where* $t \neq \mathsf{S}^m 0$, *let*

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

   then $T = VT'SUR, \tau = VT'A, \Gamma' = \Gamma_3$.
9. *If e is of the form* $\mathsf{iter}\ (\mathsf{S}^m 0)\ u\ v$, *let*

$$\begin{aligned}&&(S, B, \Gamma_0) &= \mathcal{L}(\Gamma, u)\\ &&(R, A, \Gamma_1) &= \mathcal{L}(S\Gamma_0, v)\\ &&B_0 &= RB\\ &&S_0 &= RS\\ &\textit{for } i = 1 \cdots m\{ &U_i &= \mathsf{mgu}\ A\ (B_{i-1} \multimap \beta_i);\quad \beta_i\ \textit{new}\\ &&B_i &= U_i\beta_i\\ &&S_i &= U_iS_0\\ \}&&&\\ &&\textit{Condition} &: S_1\Gamma_0 = \cdots = S_m\Gamma_0\end{aligned}$$

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

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

$$\begin{array}{ll} (R, C, \Gamma_1) & = \mathcal{L}(\Gamma, t) \\ U & = \mathsf{mgu}\ C\ \mathtt{Bool} \\ (S, \rho, \Gamma_2) & = \mathcal{L}(UR\Gamma_1, u) \\ (S', \sigma, \Gamma_3) & = \mathcal{L}(SUR\Gamma_1, v) \\ Condition & : \ \Gamma_3 = \Gamma_2 \\ V & = \mathsf{mgu}\ \sigma\ S'\rho \end{array}$$

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

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 [20]). 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 3 (Substitution). *If there is a derivation $\Gamma \mid \Gamma' \vdash_{\mathcal{L}} e : \tau$ then, for any substitution S, there is also a derivation for $S(\Gamma \mid \Gamma') \vdash_{\mathcal{L}} e : S\tau$.*

Proof. By induction over the length of the derivation.

Theorem 2 (Soundness of $\mathcal{L}$). *If $\mathcal{L}(\Gamma, e)$ succeeds with (S, τ, Γ') then there is a derivation of $S(\Gamma \mid \Gamma') \vdash_{\mathcal{L}} e : \tau$.*

Proof. By induction on the structure of terms e, using the Substitution Lemma and the fact that substitutions are idempotent. We show two 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 $R(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}} t : A$ and $S(R\Gamma_1 \mid \Gamma_2) \vdash_{\mathcal{L}} u : B$. Since Γ_2 is included in $R\Gamma_1$, and R is idempotent, also $SR(\Gamma_1 \mid \Gamma_2) \vdash u : B$. By Lemma 3 we can write the first derivation as $SR(\Gamma \mid \Gamma_1) \vdash t : SA$. Now, by $\otimes\mathsf{Intro}$ $SR(\Gamma \mid \Gamma_2) \vdash_{\mathcal{L}} e : SA \otimes B$.
2. If e is of the form $\mathsf{iter}\ t\ u\ v$ and $t \neq S^m 0$, then $\mathcal{L}(\Gamma, t)$ succeeds with (R, C, Γ_1) and $\mathsf{mgu}\ C\ \mathsf{Nat}$ 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 $\mathsf{mgu}\ B\ (T'A \multimap T'A)$ succeeds with a substitution V. Now by induction and Lemma 3 there are derivations ending in $VT'SUR(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}} t : \mathsf{Nat}$, $VT'SUR(\Gamma_1 \mid \Gamma_2) \vdash_{\mathcal{L}} u : VT'A$, and $VT'SUR(\Gamma_2 \mid \Gamma_3) \vdash_{\mathcal{L}} v : VT'A \multimap VT'A$, and the result follows by the Iter rule.

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 [8]. Note that a notion of *principal* type follows as an immediate corollary of the completeness theorem.

Theorem 3 (Completeness of $\mathcal{L}$). *If there is a derivation $S(\Gamma \mid \Gamma_1) \vdash_{\mathcal{L}} e : \tau$ for some substitution S, then:*

1. *$\mathcal{L}(\Gamma, e)$ succeeds with (R, A, Γ_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.

4 Iterative Types

In this section we present two intersection type systems closely related to System $\mathcal{L}$. The first one is based on Damas's type system[8], a less known polymorphic type system with the same power of the Hindley-Milner system. The second is based on rank-2 intersection types [4, 16].

Iterative types are a compact way of expressing several type derivations for an iterated function. Consider the iterator function itself $\lambda x.\mathsf{iter}\ t\ u\ x$. When this function is applied to a term v our type rules assume that v must be typed with every type in $It(A_0, \ldots, A_n)$. Another way to see it is to type $\mathsf{iter}\ t\ u\ x$ 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.

4.1 Polymorphic iteration

In the system presented here, which we call System $\mathcal{L}_{\mathcal{I}}$, 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 [8] in the definition of a system (later called Damas's System T [15]) 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 2. Let S range over the set of all finite non-empty subsets of *Types*. The set *Inter* 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 *Inter*. Bases that associate to term-variables elements of *Types* will be called *monomorphic*.

System $\mathcal{L}_{\mathcal{I}}$ is obtained from System $\mathcal{L}$ by replacing the rule for (Iter) by the two rules given in Figure 3.

$$\frac{\Gamma \vdash_{\mathcal{L}_\mathcal{I}} t : \mathsf{Nat} \qquad \Delta \vdash_{\mathcal{L}_\mathcal{I}} u : A_0}{\Gamma, x : \wedge It(A_0, \ldots, A_n), \Delta \vdash_{\mathcal{L}_\mathcal{I}} \mathsf{iter}\ t\ u\ x : A_n}\ (\mathsf{VarIter})$$

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

$$\frac{\Gamma, x : \wedge S \vdash_{\mathcal{L}_\mathcal{I}} \mathsf{iter}\ t\ u\ x : A \qquad \forall B_i \in S. \Delta \vdash_{\mathcal{L}_\mathcal{I}} v : B_i}{\Gamma, \Delta \vdash_{\mathcal{L}_\mathcal{I}} \mathsf{iter}\ t\ u\ v : A}\ (\mathsf{Iter})$$

Fig. 3. System $\mathcal{L}$ with Intersection Types

System $\mathcal{L}_\mathcal{I}$ 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 $\mathcal{L}_\mathcal{I}$ can be seen as a restriction of a system of rank-2 intersection types.

We now show two results relating System $\mathcal{L}$ and System $\mathcal{L}_\mathcal{I}$.

Theorem 4. *If there is a derivation $\Gamma \vdash_\mathcal{L} e : \tau$, then $\Gamma \vdash_{\mathcal{L}_\mathcal{I}} e : \tau$*

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

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

$$\Gamma, \Theta, \Delta \vdash_{\mathcal{L}_\mathcal{I}} \mathsf{iter}\ t\ u\ v : A_n.$$

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

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

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

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

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

Theorem 5. *If there is a derivation $\Gamma \vdash_{\mathcal{L}_\mathcal{I}} e : \tau$, with Γ monomorphic, then that $\Gamma \vdash_\mathcal{L} e : \tau$.*

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

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

$$\Gamma, x : \wedge It(A_0, \ldots, A_n) \vdash_{\mathcal{L}_\mathcal{I}} : \mathsf{iter}\ t\ u\ x : A_n \quad \Delta \vdash_{\mathcal{L}_\mathcal{I}} v : It(A_0, \ldots, A_n)$$

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

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

Corollary 1. $\vdash_{\mathcal{L}} e : \tau$ *iff* $\vdash_{\mathcal{L}_\mathcal{I}} e : \tau$.

4.2 Rank 2 Intersection Types: System $\mathcal{L}^2_\mathcal{I}$

Being able to type terms of the form $\mathsf{iter}\ (\mathsf{S}^m 0)\ u\ x$ using intersections like we do in System $\mathcal{L}_\mathcal{I}$, does not really give us more interesting terms, because we can not abstract on x, therefore it will never be replaced by the function to iterate. The system presented now extends System $\mathcal{L}_\mathcal{I}$ in that sense.

The rank 2 intersection type assignment for System $\mathcal{L}$ (which we call System $\mathcal{L}^2_\mathcal{I}$) is obtained from System $\mathcal{L}_\mathcal{I}$, by replacing the rules $\multimap\mathsf{Intro}$ and $\multimap\mathsf{Elim}$ by the two rules given in Figure 4. Note that we do not distinguish the types $\wedge\{A\}$

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

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

Fig. 4. Rank 2 Intersection Types version of System $\mathcal{L}$

and A. This system corresponds to a linear version of a rank 2 intersection type system with iterators, and it includes System $\mathcal{L}$ (and System $\mathcal{L}_\mathcal{I}$).

Theorem 6. *If there is a derivation* $\Gamma \vdash_{\mathcal{L}} e : \tau$*, then* $\Gamma \vdash_{\mathcal{L}^2_\mathcal{I}} e : \tau$

Proof. Similar to Theorem 4.

Note that terms typable in System $\mathcal{L}_\mathcal{I}$ are also typable in System $\mathcal{L}^2_\mathcal{I}$, since the rules $\multimap\mathsf{Intro}$ and $\multimap\mathsf{Elim}$ of System $\mathcal{L}_\mathcal{I}$, are a subcase of the same rules in System $\mathcal{L}^2_\mathcal{I}$.

System $\mathcal{L}^2_\mathcal{I}$ is stronger than System $\mathcal{L}_\mathcal{I}$ (therefore, than System $\mathcal{L}$), since it allows abstractions on polymorphic variables. Note however, that polymorphic

variables are only introduced through $\mathsf{VarIter}$. For example, we can have the following typing in System $\mathcal{L}^2_{\mathcal{I}}$:

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

but this term is not typable in System $\mathcal{L}$. Note also that System $\mathcal{L}^2_{\mathcal{I}}$ allows us to write more compact versions of admissible linear terms. Consider for example F to be a closed function with types $It(A \multimap \mathsf{Nat} \multimap \mathsf{Nat} \otimes \mathsf{Nat}, \ldots, \mathsf{Nat} \otimes \mathsf{Nat})$, then $(\lambda fp.\mathsf{cond}\ p\ (\mathsf{iter}\ (\mathsf{S}^2 0)\ 0\ f)\ (\mathsf{iter}\ (\mathsf{S}^2 0)\ (\mathsf{S}0)\ f))F$ is typable in System $\mathcal{L}^2_{\mathcal{I}}$.

Subject reduction for Systems $\mathcal{L}_{\mathcal{I}}$ and $\mathcal{L}^2_{\mathcal{I}}$ is proved in a similar way as for System $\mathcal{L}$. As for confluence, since we proved confluence for untyped terms in [3], that result, together with subject reduction, implies confluence for terms typable in Systems $\mathcal{L}_{\mathcal{I}}$ and $\mathcal{L}^2_{\mathcal{I}}$.

Summarising, we have shown how iterative types are related with intersection types, which in turn shows the expressiveness of System $\mathcal{L}$. The relation between the set of terms typable in the three systems is: $\mathcal{L} \subset \mathcal{L}_{\mathcal{I}} \subset \mathcal{L}^2_{\mathcal{I}}$. Furthermore, for closed terms (therefore programs): $\mathcal{L} = \mathcal{L}_{\mathcal{I}}$.

5 Conclusions

We have studied a new type construct, shown its relationship with intersection types, and given a type reconstruction algorithm for it. Since it is known 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] which show that System $\mathcal{L}$ can simulate Gödel's System $\mathcal{T}$, indicate that System $\mathcal{L}$ is even more expressive than System $\mathcal{T}$, it actually corresponds to a version of System $\mathcal{T}$ with a restricted form of intersection types.

References

1. S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of closed-reduction strategies. In *Proceedings of WRS 2006, 6th International Workshop on Rewriting Strategies, FLOC 2006, Seattle*, 2006.
3. S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of linear functions. In *Proceedings of Computer Science Logic, CSL 2006*, LNCS. Springer Verlag, 2006.
4. S. Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Department of Computer Science, University of Nijmegen, 1993.
5. S. Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 151(2):385–435, 1995.
6. 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.
7. 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.

8. L. M. M. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.
9. 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.
10. F. Damiani. Rank-2 intersection and polymorphic recursion, 2005.
11. 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.
12. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
13. 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.
14. 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.
15. C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.
16. T. Jim. Rank 2 type systems and recursive definitions. Technical report, Massachusetts Institute of Technology, 1995.
17. 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.
18. 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.
19. 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.
20. I. Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.
21. A. Martelli and U. Montanari. An efficient unification algorithm. *Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
22. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
23. 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.
24. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
25. 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.
26. 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.