

# Gödel's System $\mathcal{T}$ Revisited

Sandra Alves<sup>a</sup>, Maribel Fernández<sup>b</sup>,  
Mário Florido<sup>a</sup> and Ian Mackie<sup>c</sup>

<sup>a</sup> *LIACC - University of Porto,  
R. do Campo Alegre 1021/1055, 4169-007, Porto, Portugal*

<sup>b</sup> *King's College London, Department of Computer Science,  
Strand, London WC2R 2LS, U.K.*

<sup>c</sup> *LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France*

---

## Abstract

The linear lambda calculus, where variables are restricted to occur in terms exactly once, has a very weak expressive power: in particular, all functions terminate in linear time. In this paper we consider a simple extension with natural numbers and a restricted iterator: only closed linear functions can be iterated. We show properties of this linear version of Gödel's System  $\mathcal{T}$  using a closed reduction strategy, and study the class of functions that can be represented. Surprisingly, this linear calculus offers a huge increase in expressive power over previous linear versions of System  $\mathcal{T}$ , which are 'closed at construction' rather than 'closed at reduction'. We show that a linear System  $\mathcal{T}$  with closed reduction is as powerful as System  $\mathcal{T}$ .

---

## 1 Introduction

One of the many strands of work stemming from linear logic [16] is the area of linear functional programming languages (see, for instance, [1,29,23]). These languages are based on a version of the  $\lambda$ -calculus with a type system corresponding to intuitionistic linear logic, and provide explicit syntactical constructs for copying and erasing terms (corresponding to the exponentials in linear logic).

A question that arises from this work is what exactly is the computational power of a linear calculus *without* the exponentials, i.e., a calculus that is syntactically linear: all variables occur exactly once. This is a severely restricted form of the (simply typed)  $\lambda$ -calculus: computation is defined by the usual  $\beta$ -reduction rule, but since there is no duplication or erasing of terms during

reduction, this calculus has a very limited computational power — it can be shown that all functions terminate in linear time [21].

In this paper, we build this language up by introducing pairs, and natural numbers with the corresponding iterator, to obtain a linear version of Gödel's System  $\mathcal{T}$  which we shall call System  $\mathcal{L}$ . System  $\mathcal{T}$  is an extension of the simply typed  $\lambda$ -calculus with numbers and a recursion operator. It is a very simple system, yet has an enormous expressive power. We will show that its power comes essentially from primitive recursion combined with *linear* higher-order functions — we can achieve the same power in a calculus that has just these two ingredients: System  $\mathcal{L}$ .

In a correctness proof for the Geometry of Interaction [17], Girard uses a strategy for cut elimination where cut elimination steps can only take place when the exponential boxes are closed. Not only is this strategy for cut elimination simpler than the general one, it is also exceptionally efficient in terms of the number of cut elimination steps. Translations of the  $\lambda$ -calculus into linear logic inspired the work on closed reduction strategies in the  $\lambda$ -calculus [12,13]. Closed reductions avoid  $\alpha$ -conversion but (in contrast with standard weak strategies) allow reductions inside abstractions, achieving more sharing of computation. We use a closed reduction strategy in System  $\mathcal{L}$ , thus, iterators on *open* linear functions are accepted (since these terms are linear), but they are only reduced after the function becomes closed: reduction preserves linearity.

This design choice, which departs from previous linear versions of System  $\mathcal{T}$  (see for instance [11,22]), has an enormous impact in the computation power of the calculus: System  $\mathcal{L}$  is as powerful as System  $\mathcal{T}$ . Usual definitions of linear systems, which require iterator terms to be built using closed functions, are strictly less powerful than System  $\mathcal{T}$  [11,22]. We analyse the interplay between linearity, iteration and closed reduction in Gödel's System  $\mathcal{T}$  as follows:

- (1) We show that a closed reduction strategy is adequate for the evaluation of programs in System  $\mathcal{T}$ . For this, we define a version of System  $\mathcal{T}$ , called  $\mathcal{T}_c$ , where reduction can only take place when certain arguments are closed, and show that  $\mathcal{T}_c$  is equivalent to  $\mathcal{T}$ . We also define a system  $\mathcal{T}_0$  where iterators are syntactically restricted so that only closed functions can be used, and show that also  $\mathcal{T}_0$  is equivalent to  $\mathcal{T}$ . In other words, neither closed reduction nor closed construction restrictions affect System  $\mathcal{T}$ 's computational power.
- (2) We then introduce linearity constraints in System  $\mathcal{T}$ , together with a closed reduction strategy. We show that this linear version of System  $\mathcal{T}$  (System  $\mathcal{L}$ ) also has the computational power of the full System  $\mathcal{T}$ . In other words, linearity does not affect the computational power of System  $\mathcal{T}$  if we use a closed reduction strategy for iterators.
- (3) To compare the closed reduction and closed construction approaches, we

define two linear versions of System  $\mathcal{T}$  with a restricted product type. In the first one, which we call  $\mathcal{L}_0^N$ , iterator terms can only be built using closed functions. System  $\mathcal{L}_0^N$  can only encode primitive recursive functions, unlike  $\mathcal{T}_0$  and  $\mathcal{L}$  which are as powerful as System  $\mathcal{T}$ . The second system, called  $\mathcal{L}^N$ , uses a closed reduction strategy to reduce iterator terms and can encode non-primitive recursive functions, such as Ackermann's function.

**Related work.** Linearity may be defined in three main ways: syntactical, operational and denotational. Operational linearity means that redexes cannot be duplicated during evaluation (cf. *weak linear terms* in [5] and *simple terms* in [24]). Denotational linearity is achieved when only linear functions can be defined in the language [14,32] (note that denotational linear terms may use variables non-linearly). Finally, syntactical linearity requires a linear use of variables in terms, and it is the computational counterpart of linearity in linear logic. The language defined in [32] is a linear version of PCF in a denotational sense: it has a linear model (linear coherence spaces) but its terms can contain more than one occurrence of the same variable. In this paper, we present syntactically linear versions of System  $\mathcal{T}$ , where terms contain exactly one occurrence of each variable.

A number of calculi, many based on linear logic, have been designed with the aim of capturing specific complexity classes (see, for instance, [6,15,20,7,26,34,8]). *Bounded linear logic* [20] is an interesting example: it has a computational power that lies in-between the linear and the full  $\lambda$ -calculus; specifically, it captures the polynomial time computable functions. There is also previous work that uses linear types to characterise computations with time bounds [22]. Thus our work can be seen as establishing another calculus with good computational properties which does not need the full power of the exponentials, and introduces the non-linear features (copying and erasing) through alternative means.

From a categorical perspective, it is well-known that a Cartesian closed category (CCC) models the structure of the simply typed  $\lambda$ -calculus (i.e., the  $\lambda$ -calculus is the internal language for CCC [27,28]). The internal language of a symmetric monoidal closed category (SMCC) is the linear  $\lambda$ -calculus [30]. If we add a natural numbers object (NNO) to this category, then this corresponds to adding natural numbers and an iterator to the calculus. In this setting, a question that arises is therefore: what is the correspondence between CCC and SMCC+NNO? Although this is not the focus of the present paper, it is indeed a motivation for following this line of investigation.

**Overview.** In the next section, we recall the background material. Section 3 defines a version of System  $\mathcal{T}$  with closed reduction, called  $\mathcal{T}_c$ , and compares it with  $\mathcal{T}_0$ , which uses the 'closed-at-construction' approach to iteration. The main result here is that  $\mathcal{T} = \mathcal{T}_0 = \mathcal{T}_c$ . System  $\mathcal{L}$  is defined in Section 4 by imposing linearity constraints in System  $\mathcal{T}$ , together with a closed reduction strategy. In Section 5, we demonstrate that we can encode the primitive recursive functions in this calculus, and even go considerably beyond this class of functions. In Section 6, we show how to encode Gödel's System  $\mathcal{T}$ , and in Section 7 we analyse the power of linear systems with and without closed reduction. Finally we conclude the paper in Section 8. This paper is a revised and extended version of [3,4]. For more detailed proofs and examples we refer to [2].

## 2 Background

We assume the reader is familiar with the  $\lambda$ -calculus [9]. In this section we recall the main notions from Gödel's System  $\mathcal{T}$ , for more details see [19].

System  $\mathcal{T}$  is the simply typed  $\lambda$ -calculus (with arrow types, written  $A \rightarrow B$ , and products  $A \times B$ , and the usual  $\beta$ -reduction and projection rules) where a basic type  $\mathbf{N}$  for numbers and a recursor have been added. Numbers are built from 0 and  $\mathbf{S}$ ; we write  $\bar{n}$  or  $\mathbf{S}^n 0$  for  $\underbrace{\mathbf{S} \dots \mathbf{S}}_n 0$ , and in general  $t^n u$  will denote  $n$  applications of  $t$  to  $u$ . The recursor is defined by the reduction rules:

$$\begin{aligned} \mathbf{R} 0 u v &\longrightarrow u \\ \mathbf{R} (\mathbf{S} t) u v &\longrightarrow v (\mathbf{R} t u v) t \end{aligned}$$

Figure 1 shows the entire system. Typing contexts are sets of assumptions of the form  $x:A$ , where  $x$  is a variable and  $A$  is a type, such that there is at most one assumption for each variable. We denote by  $dom(\Gamma)$  the set of variables  $x_i$  such that  $x_i:A_i \in \Gamma$ , and write  $\Gamma, x:A$  to denote the "update" of  $\Gamma$  with  $x:A$ ; more precisely,  $\Gamma, x:A$  is the typing context obtained by adding the type assumption  $x:A$  in  $\Gamma$ , if  $x \notin dom(\Gamma)$ , or by replacing the type assumption for  $x$  with  $x:A$ , if  $x \in dom(\Gamma)$ .

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:

$$\begin{aligned} \text{iter } 0 u v &\longrightarrow u & \text{iter } (\mathbf{S} t) u v &\longrightarrow v(\text{iter } t u v) \end{aligned}$$

**Axiom**

$$\frac{}{\Gamma, x : A \vdash_{\mathcal{T}} x : A} \text{ (Axiom)}$$

**Logical Rules:**

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

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

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

**Numbers:**

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

$$\frac{\Gamma \vdash_{\mathcal{T}} t : \mathbb{N} \quad \Gamma \vdash_{\mathcal{T}} u : A \quad \Gamma \vdash_{\mathcal{T}} v : A \rightarrow \mathbb{N} \rightarrow A}{\Gamma \vdash_{\mathcal{T}} R t u v : A} \text{ (Rec)}$$

Fig. 1. System  $\mathcal{T}$

with the following typing rule:

$$\frac{\Gamma \vdash_{\mathcal{T}} t : \mathbb{N} \quad \Gamma \vdash_{\mathcal{T}} u : A \quad \Gamma \vdash_{\mathcal{T}} v : A \rightarrow A}{\Gamma \vdash_{\mathcal{T}} \text{iter } t u v : A} \text{ (Iter)}$$

The iterator has the same computational power as the recursor, as the following encodings show.

$$\text{iter } t u v \stackrel{\text{def}}{=} R t u (\lambda xy. vx)$$

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

In the rest of the 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).

The following properties of System  $\mathcal{T}$  will be useful in Section 5. Below  $\text{fv}(t)$  denotes the set of free variables of  $t$ .

**Lemma 1** (1) *If  $\Gamma \vdash_{\mathcal{T}} t : T$ , then  $\Gamma, x : A \vdash_{\mathcal{T}} t : T$ , for all  $x : A$  such that  $x \notin \text{dom}(\Gamma)$ .*

(2) *If  $\Gamma \vdash_{\mathcal{T}} t : T$  and  $\text{fv}(t) = \{x_1, \dots, x_n\}$  ( $n \geq 0$ ), then there exists  $A_1, \dots, A_n$  such that  $x_1:A_1, \dots, x_n:A_n \vdash_{\mathcal{T}} t : T$ . In other words, if a term is typable, then there is a type derivation using a typing context that only contains declarations for the free variables of the term.*

- (3) If  $\Gamma \vdash_{\mathcal{T}} \lambda x.u : T$  then  $T = A \rightarrow B$  and  $\Gamma, x:A \vdash_{\mathcal{T}} u : B$  for some  $A, B$ .
- (4) If  $\Gamma \vdash_{\mathcal{T}} \pi_1(s) : T$  then  $\Gamma \vdash_{\mathcal{T}} s : T \times B$  for some  $B$ .
- (5) If  $\Gamma \vdash_{\mathcal{T}} \pi_2(s) : T$  then  $\Gamma \vdash_{\mathcal{T}} s : A \times T$  for some  $A$ .

To compare the computation power of System  $\mathcal{T}$  and its restrictions, we need to define programs and their associated values.

**Definition 2** *A program in System  $\mathcal{T}$  is a closed term of base type  $\mathbf{N}$ . Values are terms of the form  $S^n 0$ ,  $\langle s, s' \rangle$ ,  $\lambda x.s$ .*

In Figure 2 we define a call-by-name evaluation strategy for System  $\mathcal{T}$ :  $t \Downarrow v$  means that the closed term  $t$  evaluates to the value  $v$ . We use the notation  $t[u/x]$  for the result of substituting  $x$  by  $u$  in  $t$ , using the standard capture-avoiding notion of substitution.

$$\begin{array}{c}
\frac{t \Downarrow \lambda x.t' \quad t'[u/x] \Downarrow v}{tu \Downarrow v} \quad \frac{t \Downarrow \langle s, s' \rangle \quad s \Downarrow v}{\pi_1(t) \Downarrow v} \quad \frac{t \Downarrow \langle s, s' \rangle \quad s' \Downarrow v}{\pi_2(t) \Downarrow v} \\
\frac{v \text{ is a value}}{v \Downarrow v} \quad \frac{t \Downarrow v}{S t \Downarrow S v} \quad \frac{t \Downarrow S^n 0 \quad s^n u \Downarrow v}{\text{iter } t u s \Downarrow v}
\end{array}$$

Fig. 2. System  $\mathcal{T}$  evaluation

**Lemma 3** *If  $t \Downarrow v$  then  $t \longrightarrow^* v$ . Moreover, if  $\Gamma \vdash_{\mathcal{T}} t : T$  and  $t$  is closed, then*

$$\begin{aligned}
T = A \times B &\Rightarrow t \Downarrow \langle u, s \rangle, \\
T = A \rightarrow B &\Rightarrow t \Downarrow \lambda x.s, \\
T = \mathbf{N} &\Rightarrow t \Downarrow S^n 0, n \geq 0.
\end{aligned}$$

**PROOF.** The first part is by induction on  $\Downarrow$ , and the second is by induction over the maximal length of derivation out of  $t$ , using the first part.  $\square$

### 3 Systems $\mathcal{T}_c$ and $\mathcal{T}_0$ : closed reduction vs. closed construction

Çağman and Hindley [10] observed that  $\alpha$ -conversion is not needed in the  $\beta$ -reduction rule of the  $\lambda$ -calculus if  $\beta$ -redexes are closed (i.e., do not contain free variables). The *closed argument* strategy defined in [12,13] is less restrictive and is also free from  $\alpha$ -conversion. It requires closed arguments in  $\beta$ -redexes:

$$(\lambda x.t)u \rightarrow t[u/x] \quad \text{if } \text{fv}(u) = \emptyset$$

When applied to System  $\mathcal{T}$ , closed reduction also restricts the application of the iteration rules: they can only be triggered if the iterated function is closed.

$$\begin{aligned} \text{iter } 0 \ u \ v &\rightarrow u && \text{if } \text{fv}(v) = \emptyset \\ \text{iter } (\text{S } t) \ u \ v &\rightarrow v(\text{iter } t \ u \ v) && \text{if } \text{fv}(v) = \emptyset \end{aligned}$$

Thus, to reduce an iterator term we must wait until the iterated function is closed. The intuition as to why this is an efficient reduction strategy is that only closed terms are copied, and can thus be fully reduced before copying. We will call System  $\mathcal{T}_c$  the version of System  $\mathcal{T}$  that uses the closed reduction strategy defined above. Although restrictive, closed reduction is still adequate:

**Theorem 4 (Adequacy of closed reduction for System  $\mathcal{T}$ )** *If  $t$  is a program, then there is a value  $v$  such that  $t \rightarrow^* v$  using closed reduction. Hence, System  $\mathcal{T}$  and System  $\mathcal{T}_c$  have the same computation power.*

**PROOF.** Since the reduction strategy is strictly included in the usual reduction, it is strongly normalising and has the subject reduction property [19]. The proof then follows the same structure as Theorem 18 that we give in detail later for the linear system.  $\square$

We now define System  $\mathcal{T}_0$ , a variant of System  $\mathcal{T}$  that uses the typing rule below for the iterator terms, that is, iterator terms must be built using closed functions:

$$\frac{\Gamma \vdash_{\mathcal{T}_0} t : \mathbf{N} \quad \Gamma \vdash_{\mathcal{T}_0} u : A \quad \vdash_{\mathcal{T}_0} v : A \rightarrow A}{\Gamma \vdash_{\mathcal{T}_0} \text{iter } t \ u \ v : A} \text{ (Iter)}$$

This constraint does not weaken the system:

**Theorem 5**  $\mathcal{T}_0 = \mathcal{T}$ .

**PROOF.** Define  $M = \lambda xy.y(xy)$ . Each iterator term  $\text{iter } n \ b \ f$  in System  $\mathcal{T}$ , where  $f$  may be an open term, is translated to the typable term  $(\text{iter } n \ (\lambda x.b) \ M)f$ , where  $x \notin \text{fv}(b)$ . It is easy to see that  $\text{iter } n \ b \ f$  and  $(\text{iter } n \ (\lambda x.b) \ M)f$  have the same normal form.  $\square$

It is worth remarking that we rely on a non-linear term  $M$  to get this result. Indeed, iterating  $M$  is essentially equivalent to constructing a Church numeral.

## 4 System $\mathcal{L}$ : a linear System $\mathcal{T}$

We define System  $\mathcal{L}$ , a linear version of System  $\mathcal{T}$ , by extending the linear  $\lambda$ -calculus [1] with numbers, pairs, and an iterator with a closed reduction strategy [13,18].

The set  $\Lambda$  of linear  $\lambda$ -terms  $t, u, \dots$  is inductively defined by:  $x \in \Lambda$ ,  $\lambda x.t \in \Lambda$  if  $x \in \text{fv}(t)$ , and  $tu \in \Lambda$  if  $\text{fv}(t) \cap \text{fv}(u) = \emptyset$ . Note that  $x$  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. Thus these conditions ensure syntactic linearity (variables occur exactly once).

In System  $\mathcal{L}$  we will also have numbers generated by 0 and S, with an iterator:

$$\text{iter } t \ u \ v \quad \text{if } \text{fv}(t) \cap \text{fv}(u) = \text{fv}(u) \cap \text{fv}(v) = \text{fv}(v) \cap \text{fv}(t) = \emptyset$$

and pairs:

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

Note that when projecting from a pair, we use both projections. A simple example of such a term is the function that swaps the components of a pair:

$$\lambda x. \text{let } \langle y, z \rangle = x \text{ in } \langle z, y \rangle$$

Tuples of any size can be built from pairs. As an example,  $\langle x_1, x_2, x_3 \rangle = \langle x_1, \langle x_2, x_3 \rangle \rangle$  and  $\text{let } \langle x_1, x_2, x_3 \rangle = u \text{ in } t$  represents the term

$$\text{let } \langle x_1, y \rangle = u \text{ in let } \langle x_2, x_3 \rangle = y \text{ in } t$$

Table 1 summarises the syntax of System  $\mathcal{L}$ . The set of terms given by Table 1 is called  $\Lambda_{\mathcal{L}}$ . Note that  $\lambda$  and **let** are binders; we work with terms modulo  $\alpha$ -conversion as usual.

**Definition 6 (Closed reduction)** *The reduction rules for System  $\mathcal{L}$  are given in Table 2. Substitution is a meta-operation defined as usual, and reductions can take place in any context. We use the same symbol to denote the reduction relation in System  $\mathcal{L}$  and in System  $\mathcal{T}$  since the intended relation will always be clear from the context.*

Normal forms are not the same as in the  $\lambda$ -calculus: for example,  $\lambda x.(\lambda y.y)x$  is a normal form. Note that all the substitutions created during reduction (rules *Beta* and *Let*) are closed, and the *Iter* rules are only triggered when the function  $v$  is closed.

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

Table 1  
Terms in System  $\mathcal{L}$

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

Table 2  
Closed reduction

**Lemma 7 (Correctness of substitution)** *Let  $t$  and  $u$  be System  $\mathcal{L}$  terms. If  $z \in \text{fv}(t)$  and  $\text{fv}(u) = \emptyset$ , then  $t[u/z]$  is also a System  $\mathcal{L}$  term. More generally, if  $\text{fv}(t) \cap \text{fv}(u) = \emptyset$  then  $t[u/z]$  satisfies the variable constraints.*

**PROOF.** Straightforward induction on the structure of  $t$ . □

**Lemma 8 (Correctness of reduction)** *Let  $t$  be a System  $\mathcal{L}$  term such that  $t \longrightarrow u$ , then  $\text{fv}(t) = \text{fv}(u)$  and  $u$  is also a System  $\mathcal{L}$  term.* □

**PROOF.** By simultaneous induction, showing that all the rewrite rules preserve the variable constraints given in Table 1.

Note that reduction preserves the free variables of the term, but the free variables of a subterm may change. In particular, a subterm of the form iter  $n u v$

may become closed after a reduction in a superterm, triggering in this way a reduction with an *Iter* rule.

Although linear, System  $\mathcal{L}$  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:  $\Omega \longrightarrow^* \Omega$ . In the remainder of this section we define a linear type system for System  $\mathcal{L}$  and show that typable terms are strongly normalisable.

#### 4.1 Types for System $\mathcal{L}$

The set of *linear types* is generated by the grammar:

$$A, B ::= \mathbf{N} \mid A \multimap B \mid A \otimes B$$

##### Axiom

$$\frac{}{x : A \vdash_{\mathcal{L}} x : A} \text{ (Axiom)}$$

##### Logical Rules

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

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

##### Numbers

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

$$\frac{\Gamma \vdash_{\mathcal{L}} t : \mathbf{N} \quad \Theta \vdash_{\mathcal{L}} u : A \quad \Delta \vdash_{\mathcal{L}} v : A \multimap A}{\Gamma, \Theta, \Delta \vdash_{\mathcal{L}} \text{iter } t u v : A} \text{ (Iter)}$$

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

We associate types to terms in System  $\mathcal{L}$  using the typing rules given in Figure 3. We use a Curry-style type system; the typing rules specify how to assign types to untyped terms (there are no type decorations). Again, typing contexts are sets of type assumptions of the form  $x : A$ , and  $\text{dom}(\Gamma)$  denotes the set of variables such that  $x_i : A_i \in \Gamma$ . Note that the axiom has exactly one type assumption in the context, and we do not have weakening and contraction rules — we are in a linear system. For the same reason, the logical rules split the context between the premises. The rules for numbers are standard. In the case of a term of the form  $\text{iter } t u v$ , we check that  $t$  is a term of type  $\mathbf{N}$  and that  $v$  and  $u$  are compatible.

Since we are in a linear system, we have:

**Lemma 9** *If  $\Gamma \vdash_{\mathcal{L}} t : A$  then  $\text{dom}(\Gamma) = \text{fv}(t)$ .*

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

In order to prove that reduction preserves types we use a substitution lemma.

**Lemma 10 (Substitution)** *If  $\Gamma, x : A \vdash_{\mathcal{L}} t : B$  and  $\Delta \vdash_{\mathcal{L}} u : A$ , where  $\text{fv}(t) \cap \text{fv}(u) = \emptyset$ , then  $\Gamma, \Delta \vdash_{\mathcal{L}} t[u/x] : B$ .*

**PROOF.** By induction on the type derivation  $\Gamma, x : A \vdash_{\mathcal{L}} t : B$ . Note that, by Lemma 9,  $x \in \text{fv}(t)$ . □

**Theorem 11 (Subject Reduction)** *If  $\Gamma \vdash_{\mathcal{L}} M : A$  and  $M \longrightarrow N$ , then  $\Gamma \vdash_{\mathcal{L}} N : A$ .*

**PROOF.** By induction on the type derivation  $\Gamma \vdash_{\mathcal{L}} M : A$ , using Lemma 9. The Substitution Lemma 10 is used in the cases of application and pairs. □

## 4.2 Strong normalisation

In System  $\mathcal{L}$ , every sequence of reductions starting from a typable term is finite. To prove it, we define a translation from System  $\mathcal{L}$  into System  $\mathcal{T}$ , and use the strong normalisation property of System  $\mathcal{T}$ .

**Definition 12** *Types and terms in System  $\mathcal{L}$  are compiled into System  $\mathcal{T}$ , denoted by  $J\cdot K$ , in the direct way, except for let constructs which use projections:*

$$\begin{aligned}
JN K &= N & JA \multimap BK &= JAK \rightarrow JBK & JA \otimes BK &= JAK \times JBK \\
J0 K &= 0 & Jx K &= x \\
Jtu K &= JtK JuK & J\lambda y. t K &= \lambda y. JtK \\
JS t K &= S JtK & J\langle t, u \rangle K &= \langle JtK, JuK \rangle \\
J\text{let } \langle x, y \rangle = t \text{ in } u K &= JuK[(\pi_1 JtK)/x][(\pi_2 JtK)/y] \\
J\text{iter } t \ u \ v K &= \text{iter } JtK \ JuK \ JvK
\end{aligned}$$

Also, if  $\Gamma = x_1 : A_1, \dots, x_n : A_n$ , then  $J\Gamma K = x_1 : JA_1 K, \dots, x_n : JA_n K$ .

To simulate reductions we use the following properties, which are proved by routine induction.

**Lemma 13** (1) Let  $t, u$  be terms in System  $\mathcal{L}$  such that  $\text{fv}(u) \cap \text{fv}(t) = \emptyset$  and  $x \in \text{fv}(t)$ . Then  $JtK[J uK/x] = Jt[u/x]K$ .  
(2) If  $\Gamma \vdash_{\mathcal{L}} t : T$ , then  $J\Gamma K \vdash_{\mathcal{T}} JtK : JTK$ .

**Lemma 14** If  $t \longrightarrow t'$ , then  $JtK \longrightarrow^+ Jt'K$ .

**PROOF.** By induction on  $t$ . The only interesting cases are when  $t$  is an application or a let construct, and reduction takes place at the root position. In both cases the result follows from Lemma 13, part 1, because substitutions are closed. We show the diagram for the case of a let construct.

$$\begin{array}{ccc}
\text{let } \langle x, y \rangle = \langle a, b \rangle \text{ in } u & \longrightarrow & u[a/x][b/y] \\
\downarrow \text{J}\cdot\text{K} & & \downarrow \text{J}\cdot\text{K}(\text{Lemma 13, 1}) \\
\text{JuK}[\pi_1 \text{J}\langle a, b \rangle \text{K}/x][\pi_2 \text{J}\langle a, b \rangle \text{K}/y] & \xrightarrow{*} & \text{JuK}[\text{JaK}/x][\text{JbK}/y]
\end{array}$$

□

**Corollary 15** If  $JtK$  is strongly normalisable, so is  $t$ .

**Theorem 16 (Strong normalisation)** If  $\Gamma \vdash_{\mathcal{L}} t : T$ , then  $t$  is strongly normalisable.

**PROOF.** Consequence of Lemma 13, SN of System  $\mathcal{T}$  and Corollary 15. □

### 4.3 Confluence and adequacy

System  $\mathcal{L}$  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 [31]). In fact, all System  $\mathcal{L}$  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 17 (Confluence)**  $\longrightarrow$  is Church-Rosser.

**Theorem 18 (Adequacy)** *If  $t$  is a closed and typable System  $\mathcal{L}$  term, then one of the following holds:*

- $\vdash_{\mathcal{L}} t : \mathbf{N}$  and  $t \longrightarrow^* \bar{n}$
- $\vdash_{\mathcal{L}} t : A \multimap B$  and  $t \longrightarrow^* \lambda x.u$  for some term  $u$ .
- $\vdash_{\mathcal{L}} t : A \otimes B$  and  $t \longrightarrow^* \langle u, v \rangle$  for some terms  $u, v$ .

**PROOF.** By Lemma 9, typing judgements for  $t$  have the form  $\vdash_{\mathcal{L}} t : T$ , where  $T$  is  $\mathbf{N}$ ,  $A \multimap B$  or  $A \otimes B$ . By Subject Reduction, Strong Normalisation, and Lemma 8,  $t$  has a closed normal form  $u$  of the same type. Thus, it is sufficient to prove that if  $u$  is a normal form then one of the following holds:

- $\vdash_{\mathcal{L}} u : \mathbf{N}$  and  $u = \bar{n}$
- $\vdash_{\mathcal{L}} u : A \multimap B$  and  $u = \lambda x.s$  for some term  $s$ .
- $\vdash_{\mathcal{L}} u : A \otimes B$  and  $u = \langle a, b \rangle$  for some terms  $a, b$ .

We proceed by induction on  $u$ . We show the case when  $\vdash_{\mathcal{L}} u : \mathbf{N}$  (the others are similar). In this case  $u$  can only be an application, a let construct, an iterator or a number. Below we show the case of an iterator.

Assume  $u = \text{iter } n \ s \ v$ . Since  $u$  is closed, so are  $n, t$  and  $v$ . Since  $u$  is typable  $n$  must be a term of type  $\mathbf{N}$ , and by induction,  $n$  is a number. But then the *Iter* rule applies (contradiction).

The cases of application and let are similar. The only case that does not lead to a contradiction is a number.  $\square$

## 5 Primitive recursive functions linearly

In this section, we show that although System  $\mathcal{L}$  is linear it is possible to erase and copy numbers. Using these operations, we can define the primitive recursive functions and also go beyond by encoding Ackermann's function.

**Erasing linearly.** The projection functions `fst`, `snd` defined below are typable (see the type derivation for `fst :  $\mathbf{N} \otimes \mathbf{N} \multimap \mathbf{N}$`  in Figure 4).

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

**Lemma 19** *For any numbers  $a$  and  $b$ ,  $\text{fst}\langle \bar{a}, \bar{b} \rangle \longrightarrow^* \bar{a}$  and  $\text{snd}\langle \bar{a}, \bar{b} \rangle \longrightarrow^* \bar{b}$ .*

$$\frac{\frac{\frac{x : \mathbf{N} \otimes \mathbf{N} \vdash_{\mathcal{L}} x : \mathbf{N} \otimes \mathbf{N}}{\quad} \quad \frac{\frac{v : \mathbf{N} \vdash_{\mathcal{L}} v : \mathbf{N}}{\quad} \quad \frac{u : \mathbf{N} \vdash_{\mathcal{L}} u : \mathbf{N}}{\quad} \quad \frac{z : \mathbf{N} \vdash_{\mathcal{L}} z : \mathbf{N}}{\quad}}{\vdash_{\mathcal{L}} \lambda z. z : \mathbf{N} \multimap \mathbf{N}}}{\quad}}{\frac{x : \mathbf{N} \otimes \mathbf{N} \vdash_{\mathcal{L}} \mathbf{let} \langle u, v \rangle = x \mathbf{in} \mathbf{iter} v u (\lambda z. z) : \mathbf{N}}{\vdash_{\mathcal{L}} \lambda x. \mathbf{let} \langle u, v \rangle = x \mathbf{in} \mathbf{iter} v u (\lambda z. z) : (\mathbf{N} \otimes \mathbf{N}) \multimap \mathbf{N}}}$$

Fig. 4. Typing the projections

**PROOF.** We show the case for `fst`. Let  $\bar{a} = S^n 0$ ,  $\bar{b} = S^m 0$ .

$$\begin{aligned}
\mathbf{fst} \langle \bar{a}, \bar{b} \rangle &= (\lambda x. \mathbf{let} \langle u, v \rangle = x \mathbf{in} \mathbf{iter} v u (\lambda z. z)) \langle S^n 0, S^m 0 \rangle \\
&\longrightarrow (\mathbf{let} \langle u, v \rangle = \langle S^n 0, S^m 0 \rangle \mathbf{in} \mathbf{iter} v u \lambda z. z) \\
&\longrightarrow \mathbf{iter} (S^m 0) (S^n 0) (\lambda z. z) \\
&\longrightarrow^* (\lambda z. z)^m (S^n 0) \longrightarrow S^n 0 = \bar{a} \quad \square
\end{aligned}$$

We do not claim that this is the only way of encoding the projections in System  $\mathcal{L}$ , indeed there are several ways of erasing numbers. For instance, we could define `fst` =  $\lambda x. \mathbf{let} \langle u, v \rangle = x \mathbf{in} \mathbf{iter} 0 u \lambda x. \mathbf{iter} x v \lambda y. y$ , relying on the rule `iter 0 u v`  $\rightarrow$   $u$  (which erases  $v$ ). However, the technique of erasing a number 'by consuming it' has some advantages: we will see in the next section that we can indeed erase any term in this way, and thus the rule `iter 0 u v`  $\rightarrow$   $u$  could be replaced by a linear rule using this technique.

**Copying linearly.** The following function  $C : \mathbf{N} \multimap \mathbf{N} \otimes \mathbf{N}$  can be used to copy numbers:

$$C = \lambda x. \mathbf{iter} x \langle 0, 0 \rangle (\lambda x. \mathbf{let} \langle a, b \rangle = x \mathbf{in} \langle Sa, Sb \rangle)$$

**Lemma 20** *For any number  $n$ ,  $C \bar{n} \longrightarrow^* \langle \bar{n}, \bar{n} \rangle$ .*

**PROOF.** By induction on  $\bar{n}$ .

$$C 0 \longrightarrow \mathbf{iter} 0 \langle 0, 0 \rangle (\lambda x. \mathbf{let} \langle a, b \rangle = x \mathbf{in} \langle Sa, Sb \rangle) \longrightarrow \langle 0, 0 \rangle$$

$$\begin{aligned}
C(\mathbf{S}^{t+1} 0) &= \text{iter}(\mathbf{S}^{t+1} 0) \langle 0, 0 \rangle (\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle) \\
&\longrightarrow (\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle) \\
&\quad (\text{iter}(\mathbf{S}^t 0) \langle 0, 0 \rangle (\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle)) \\
&\longrightarrow^* (\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle) \langle \bar{t}, \bar{t} \rangle \\
&\longrightarrow \text{let} \langle a, b \rangle = \langle \bar{t}, \bar{t} \rangle \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle \longrightarrow \langle \mathbf{S}\bar{t}, \mathbf{S}\bar{t} \rangle \quad \square
\end{aligned}$$

We give below the type derivation for  $C$  (where  $F$  is the subterm  $(\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle)$ ):

$$\frac{\frac{\frac{}{a : \mathbf{N} \vdash_{\mathcal{L}} a : \mathbf{N}} \quad \frac{}{b : \mathbf{N} \vdash_{\mathcal{L}} b : \mathbf{N}}}{a : \mathbf{N} \vdash_{\mathcal{L}} \mathbf{S}a : \mathbf{N} \quad b : \mathbf{N} \vdash_{\mathcal{L}} \mathbf{S}b : \mathbf{N}}}{x : \mathbf{N} \otimes \mathbf{N} \vdash_{\mathcal{L}} x : \mathbf{N} \otimes \mathbf{N} \quad a : \mathbf{N}, b : \mathbf{N} \vdash_{\mathcal{L}} \langle \mathbf{S}a, \mathbf{S}b \rangle : \mathbf{N} \otimes \mathbf{N}}}{x : \mathbf{N} \otimes \mathbf{N} \vdash_{\mathcal{L}} \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle : \mathbf{N} \otimes \mathbf{N}}}{\vdash_{\mathcal{L}} (\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle) : (\mathbf{N} \otimes \mathbf{N}) \multimap (\mathbf{N} \otimes \mathbf{N})}$$

$$\frac{\frac{\frac{}{x : \mathbf{N} \vdash_{\mathcal{L}} x : \mathbf{N}} \quad \frac{\frac{}{\vdash_{\mathcal{L}} 0 : \mathbf{N}} \quad \frac{}{\vdash_{\mathcal{L}} 0 : \mathbf{N}}}{\vdash_{\mathcal{L}} \langle 0, 0 \rangle : \mathbf{N} \otimes \mathbf{N}}}{x : \mathbf{N} \vdash_{\mathcal{L}} \text{iter } x \langle 0, 0 \rangle (\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle) : \mathbf{N} \otimes \mathbf{N}}}{\vdash_{\mathcal{L}} \lambda x. \text{iter } x \langle 0, 0 \rangle (\lambda x. \text{let} \langle a, b \rangle = x \text{ in} \langle \mathbf{S}a, \mathbf{S}b \rangle) : \mathbf{N} \multimap (\mathbf{N} \otimes \mathbf{N})}$$

This technique can be applied to other data structures (e.g. linear lists). More interestingly, we will show in Section 6 that iterators allow us to copy and erase any closed term.

**Examples.** The following arithmetic functions can be written in System  $\mathcal{L}$  (we omit the type derivations).

- $\text{add} = \lambda mn. \text{iter } m \ n \ \text{succ}$ , where  $\text{succ} = \lambda x. \mathbf{S}x$
- $\text{mult} = \lambda mn. \text{iter } m \ 0 \ (\text{add } n)$
- $\text{exp} = \lambda mn. \text{iter } n \ (\mathbf{S} \ 0) \ (\text{mult } m)$
- $\text{pred} = \lambda n. \text{fst}(\text{iter } n \ \langle 0, 0 \rangle (\lambda x. \text{let} \langle t, u \rangle = C(\text{snd } x) \text{ in} \langle t, \mathbf{S} \ u \rangle))$
- $\text{is\_zero} = \lambda n. \text{fst}(\text{iter } n \ \langle 0, \mathbf{S} \ 0 \rangle (\lambda x. C(\text{snd } x)))$
- $\text{sub} = \lambda mn. \text{iter } n \ m \ \text{pred}$
- $\text{fact} = \lambda n. \text{snd}(\text{iter } n \ \langle \bar{0}, \bar{1} \rangle (\lambda x. \text{let} \langle t, u \rangle = x \text{ in} \text{let} \langle t_1, t_2 \rangle = C \ t \ \text{in} \ F))$   
where  $F = \langle \mathbf{S} \ t_1, \text{mult } u \ (\mathbf{S} \ t_2) \rangle$ .

**Primitive recursive functions.** A function  $f : \mathbf{N}^n \rightarrow \mathbf{N}$  is primitive recursive if it can be defined using the natural numbers, projections

$$\pi_i^n(x_1, \dots, x_n) = x_i, \quad 1 \leq i \leq n$$

(we omit the superindex when there is no ambiguity), composition of functions, and the primitive recursive scheme, which allows us to define a recursive function  $h$  using two auxiliary primitive recursive functions  $f$  and  $g$ :

$$\begin{aligned} h(\vec{x}, 0) &= f(\vec{x}) \\ h(\vec{x}, \mathbf{S} n) &= g(\vec{x}, h(\vec{x}, n), n) \end{aligned}$$

The notation  $\vec{x}$  is used as abbreviation for a sequence  $x_1, \dots, x_m$ . Note that in the last equation, the numbers  $\vec{x}$  and  $n$  are copied.

System  $\mathcal{L}$  can express the whole class of primitive recursive functions. We have already shown we can project, and of course we have composition. We now show how to encode a function  $h$  defined by primitive recursion from  $f$  and  $g$ . For simplicity we assume  $h$  is a binary function ( $h : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ ).

First, assume  $h$  is defined by the following, simpler scheme (it uses  $n$  only once in the second equation):

$$\begin{aligned} h(x, 0) &= f(x) \\ h(x, n + 1) &= g(x, h(x, n)) \end{aligned}$$

Assume that there are closed terms in System  $\mathcal{L}$  representing the functions  $f$  and  $g$ ; we will denote them by  $f$  and  $g$  since there is no ambiguity. Using  $f : \mathbf{N} \multimap \mathbf{N}$  and  $g : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$ , the function  $h$  can be defined by the typable term  $(\text{iter } n f g')x : \mathbf{N}$ , where  $g'$  is the term:

$$\lambda y. \lambda z. \text{let } \langle z_1, z_2 \rangle = C z \text{ in } g z_1 (y z_2) : (\mathbf{N} \multimap \mathbf{N}) \multimap (\mathbf{N} \multimap \mathbf{N})$$

Indeed, we can show by induction that  $(\text{iter } n f g')x$ , where  $x$  and  $n$  are numbers, reduces to the number  $h(x, n)$ , using Lemma 20 to copy numbers.

Now to encode the standard primitive recursive scheme, which has an extra  $n$  in the last equation, all we need to do is copy  $n$ . We use the notation introduced above ( $f, g$  are auxiliary functions, but  $g$  has now the type  $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$ ):

$$h = \lambda x n. \text{let } \langle n_1, n_2 \rangle = C n \text{ in } s (\text{pred } n_1) x$$

where  $\text{pred}$  is the predecessor function defined above and

$$s = \text{iter } n_2 (\lambda x_1. (\text{iter } x_1 I I) f) s'$$

$$s' = \lambda y x_2 z.$$

$$\text{let } \langle z_1, z_2 \rangle = C z \text{ in } (\text{let } \langle w_1, w_2 \rangle = C x_2 \text{ in } g z_1 (y (\text{pred } w_1) z_2) w_2)$$

**Beyond primitive recursion.** Ackermann's function is a standard example of a non primitive recursive function:

$$\text{ack}(0, n) = \mathbf{S } n$$

$$\text{ack}(\mathbf{S } n, 0) = \text{ack}(n, \mathbf{S } 0)$$

$$\text{ack}(\mathbf{S } n, \mathbf{S } m) = \text{ack}(n, \text{ack}(\mathbf{S } n, m))$$

In a higher-order functional language, there is an alternative definition. Let  $\text{succ} = \lambda x. \mathbf{S } x : \mathbf{N} \multimap \mathbf{N}$ , then  $\text{ack}(m, n) = a m n$ , where  $a$  is defined by:

$$a 0 = \text{succ} \quad A g 0 = g(\mathbf{S } 0)$$

$$a (\mathbf{S } n) = A (a n) \quad A g (\mathbf{S } n) = g(A g n)$$

We can define  $a$  and  $A$  in System  $\mathcal{L}$  as follows:

$$a = \lambda n. \text{iter } n \text{ succ } A : \mathbf{N} \multimap (\mathbf{N} \multimap \mathbf{N})$$

$$A = \lambda g n. \text{iter } (\mathbf{S } n) (\mathbf{S } 0) g : (\mathbf{N} \multimap \mathbf{N}) \multimap \mathbf{N} \multimap \mathbf{N}$$

We show by induction that this encoding is correct:

- $a 0 \longrightarrow \text{iter } 0 \text{ succ } A \longrightarrow \text{succ}$   
 $A g 0 \longrightarrow^* \text{iter } (\mathbf{S } 0) (\mathbf{S } 0) g \longrightarrow^* g(\mathbf{S } 0)$
- $a (\mathbf{S } n) \longrightarrow \text{iter } (\mathbf{S}^n 0) \text{ succ } A \longrightarrow A(\text{iter } n \text{ succ } A) \longrightarrow^* A(a n)$   
 $A g (\mathbf{S } n) \longrightarrow^* \text{iter } (\mathbf{S}(\mathbf{S } n)) (\mathbf{S } 0) g \longrightarrow g(\text{iter } (\mathbf{S } n) (\mathbf{S } 0) g) \longrightarrow^* g(A g n).$

Then Ackermann's function can be defined in System  $\mathcal{L}$  by the typable term:

$$\text{ack} = \lambda m n. (\text{iter } m \text{ succ } (\lambda g u. \text{iter } (\mathbf{S } u) (\mathbf{S } 0) g)) n : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$$

Note that  $\text{iter } (\mathbf{S } u) (\mathbf{S } 0) g$  cannot be typed in the linear system defined in [11], because  $g$  is a free variable. We allow building the term with the free variable  $g$ , but we do not allow reduction until it is closed.

## 6 The power of System $\mathcal{L}$

In this section, we show how to compile System  $\mathcal{T}$  programs into System  $\mathcal{L}$ ; i.e., we show that System  $\mathcal{L}$  has all the computation power of System  $\mathcal{T}$ .

**Explicit erasing.** In the linear  $\lambda$ -calculus, we are not able to erase arguments. However, terms are consumed by reduction. The idea of erasing by consuming is not new, it is related to *solvability* (see [9] for instance). Our goal in this section is to give an analogous result that allows us to obtain a general form of erasing.

**Definition 21 (Erasing)** *We define the following mutually recursive operations  $\mathcal{E}$  and  $\mathcal{M}$ , which, respectively, erase and create a System  $\mathcal{L}$  term. If  $\Gamma \vdash_{\mathcal{L}} t : T$ , then  $\mathcal{E}(t, T)$  is defined as follows (where  $I = \lambda x.x$ ):*

$$\begin{aligned} \mathcal{E}(t, \mathbf{N}) &= \text{iter } t \ I \ I & \mathcal{M}(\mathbf{N}) &= 0 \\ \mathcal{E}(t, A \multimap B) &= \mathcal{E}(t\mathcal{M}(A), B) & \mathcal{M}(A \multimap B) &= \lambda x.\mathcal{E}(x, A)\mathcal{M}(B) \\ \mathcal{E}(t, A \otimes B) &= \text{let } \langle x, y \rangle = t \text{ in } \mathcal{E}(x, A)\mathcal{E}(y, B) & \mathcal{M}(A \otimes B) &= \langle \mathcal{M}(A), \mathcal{M}(B) \rangle \end{aligned}$$

**Lemma 22** *If  $\Gamma \vdash_{\mathcal{L}} t : T$  then:*

- (1)  $\text{fv}(\mathcal{E}(t, T)) = \text{fv}(t)$  and  $\Gamma \vdash_{\mathcal{L}} \mathcal{E}(t, T) : A \multimap A$ , for any  $A$ .
- (2)  $\mathcal{M}(T)$  is a closed System  $\mathcal{L}$  term such that  $\vdash_{\mathcal{L}} \mathcal{M}(T) : T$ .

**PROOF.** Simultaneous induction on  $T$ .

$T = \mathbf{N}$ :

- $\text{fv}(\mathcal{E}(t, \mathbf{N})) = \text{fv}(\text{iter } t \ I \ I) = \text{fv}(t)$ , and  $\Gamma \vdash_{\mathcal{L}} \text{iter } t \ I \ I : A \multimap A$ , for any  $A$ .
- $\text{fv}(\mathcal{M}(\mathbf{N})) = \text{fv}(0) = \emptyset$ , and  $\vdash_{\mathcal{L}} 0 : \mathbf{N}$ .

$T = A \otimes B$ :

- $\text{fv}(\mathcal{E}(t, A \otimes B)) = \text{fv}(\text{let } \langle x, y \rangle = t \text{ in } \mathcal{E}(x, A)\mathcal{E}(y, B)) = \text{fv}(t)$ . By induction:  $x : A \vdash_{\mathcal{L}} \mathcal{E}(x, A) : (C \multimap C) \multimap (C \multimap C)$ ,  $y : B \vdash_{\mathcal{L}} \mathcal{E}(y, B) : C \multimap C$ , then  $x : A, y : B \vdash_{\mathcal{L}} \mathcal{E}(x, A)\mathcal{E}(y, B) : C \multimap C$ . Then  $\Gamma \vdash_{\mathcal{L}} \mathcal{E}(t, A \otimes B) : C \multimap C$ , for any  $C$ .
- $\text{fv}(\mathcal{M}(A \otimes B)) = \text{fv}(\langle \mathcal{M}(A), \mathcal{M}(B) \rangle) = \emptyset$  by IH(2), and  $\vdash_{\mathcal{L}} \langle \mathcal{M}(A), \mathcal{M}(B) \rangle : A \otimes B$  by IH(2).

$T = A \multimap B$ :

- $\text{fv}(\mathcal{E}(t, A \multimap B)) = \text{fv}(\mathcal{E}(t\mathcal{M}(A), B)) = \text{fv}(t\mathcal{M}(A)) = \text{fv}(t)$  by IH (1 and 2). Also, by IH(1)  $\Gamma \vdash_{\mathcal{L}} \mathcal{E}(t\mathcal{M}(A), B) : C \multimap C$  for any  $C$ , since  $\vdash_{\mathcal{L}} \mathcal{M}(A) : A$  by IH(2).
- $\text{fv}(\mathcal{M}(A \multimap B)) = \text{fv}(\lambda x. \mathcal{E}(x, A)\mathcal{M}(B)) = \emptyset$  by IH(1 and 2). Also,  $\vdash_{\mathcal{L}} \mathcal{M}(A \multimap B) : A \multimap B$  because by IH(1)  $x : A \vdash_{\mathcal{L}} \mathcal{E}(x, A) : B \multimap B$  and by IH(2)  $\vdash_{\mathcal{L}} \mathcal{M}(B) : B$ .  $\square$

**Lemma 23** *If  $x : A \vdash_{\mathcal{L}} t : T$  and  $\vdash_{\mathcal{L}} v : A$  then:  $\mathcal{E}(t, T)[v/x] = \mathcal{E}(t[v/x], T)$ .*

**PROOF.** By induction on  $T$ , using the fact that  $\vdash_{\mathcal{L}} t[v/x] : T$ .  $\square$

**Lemma 24 (Erase)** *If  $\vdash_{\mathcal{L}} t : T$  (i.e.  $\text{fv}(t) = \emptyset$ ) then  $\mathcal{E}(t, T) \longrightarrow^* I$ .*

**PROOF.** By induction on  $T$ , using Theorem 18:

$$\mathcal{E}(t, \mathbf{N}) = \text{iter } t \ I \ I \longrightarrow^* \text{iter } (\mathbf{S}^n 0) \ I \ I \longrightarrow^* I$$

If  $T = A \otimes B$ , then  $t \longrightarrow^* \langle a, b \rangle$  and by Theorem 11 and Lemma 8  $\vdash_{\mathcal{L}} a : A$  and  $\vdash_{\mathcal{L}} b : B$ . By induction,  $\mathcal{E}(a, A) \longrightarrow^* I$  and  $\mathcal{E}(b, B) \longrightarrow^* I$ , therefore  $\text{let } \langle x, y \rangle = \langle a, b \rangle \text{ in } \mathcal{E}(x, A)\mathcal{E}(y, B) \longrightarrow^* I$ .

If  $T = A \multimap B$  then  $\mathcal{E}(t, A \multimap B) = \mathcal{E}(t\mathcal{M}(A), B)$ . By Lemma 22  $\mathcal{M}(A)$  is a closed System  $\mathcal{L}$  term of type  $A$ , thus by induction  $\mathcal{E}(t\mathcal{M}(A), B) \longrightarrow^* I$ .  $\square$

**Explicit copying.** We have shown how to duplicate numbers in Section 5, but to simulate System  $\mathcal{T}$  we need to be able to copy arbitrary terms. The previous technique can be generalised to other data structures, but not to functions. However, the iterator copies (closed) functions. Our aim now is to harness this.

**Lemma 25 (Duplication)** *For each type  $A$ , there is a System  $\mathcal{L}$  term  $D^A : A \multimap A \otimes A$ , such that  $D^A t \longrightarrow^* \langle t, t \rangle$ , for any closed System  $\mathcal{L}$  term  $t$  of type  $A$ .*

**PROOF.** Define  $D^A : A \multimap A \otimes A$  as:

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

Now it is straightforward to show that:

$$\begin{aligned}
D^A t &\longrightarrow \text{iter } (\mathbb{S}^2 0) \langle \mathcal{M}(A), \mathcal{M}(A) \rangle (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle) \\
&\longrightarrow^* (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle)^2 \langle \mathcal{M}(A), \mathcal{M}(A) \rangle \\
&\longrightarrow^* (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle) (\mathcal{E}(\mathcal{M}(A), A) \langle \mathcal{M}(A), t \rangle) \\
&\longrightarrow^* (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle) \langle \mathcal{M}(A), t \rangle \\
&\longrightarrow^* \mathcal{E}(\mathcal{M}(A), A) \langle t, t \rangle \longrightarrow^* \langle t, t \rangle \quad \square
\end{aligned}$$

A function of type  $A \otimes A \multimap A \otimes A$  is iterated, and this idea can also scale up, e.g.:  $D_3^A : A \multimap A \otimes A \otimes A$ . The base will be  $\langle \mathcal{M}(A), \mathcal{M}(A), \mathcal{M}(A) \rangle$ , and iterate a function  $A^3 \multimap A^3$ . This result also applies to numbers, so we have two different ways of copying numbers in System  $\mathcal{L}$ .

**Definition 26 (Duplicator)** For  $n > 1$ , we define a generalised duplicator  $D_n^A : A \multimap \underbrace{A \otimes \dots \otimes A}_n$ , as:

$$\lambda x. \text{iter } (\mathbb{S}^n 0) \langle \mathcal{M}(A), \dots, \mathcal{M}(A) \rangle (\lambda y. \text{let } \langle x_1, \dots, x_n \rangle = y \text{ in } \mathcal{E}(x_1, A) \langle x_2, \dots, x_n, x \rangle)$$

## 6.1 Compilation

We now put the previous ideas together to give a formal compilation of System  $\mathcal{T}$  into System  $\mathcal{L}$ .

**Definition 27** System  $\mathcal{T}$  types are translated into System  $\mathcal{L}$  types using  $\langle \cdot \rangle$  defined by:

$$\langle \mathbb{N} \rangle = \mathbb{N} \quad \langle A \rightarrow B \rangle = \langle A \rangle \multimap \langle B \rangle \quad \langle A \times B \rangle = \langle A \rangle \otimes \langle B \rangle$$

If  $\Gamma = x_1 : T_1, \dots, x_n : T_n$  then  $\langle \Gamma \rangle = x_1 : \langle T_1 \rangle, \dots, x_n : \langle T_n \rangle$ .

We will now translate typable terms from System  $\mathcal{T}$  into typable System  $\mathcal{L}$  terms. Recall that by Lemma 1, part 2, if  $t$  is typable in System  $\mathcal{T}$  then there exists a type derivation  $\Gamma \vdash_{\mathcal{T}} t : T$  where  $\text{dom}(\Gamma) = \text{fv}(t)$ .

In the remainder of this paper, for convenience, we make the following abbreviations:

$$\begin{aligned}
C_{x:A}^{x_1, \dots, x_n} t &= \text{let } \langle x_1, \dots, x_n \rangle = D_n^A x \text{ in } t \\
A_y^x t &= ([x]t)[y/x], \quad \text{where } [x]t \text{ is defined below.}
\end{aligned}$$

**Definition 28 (Compilation)** Let  $t$  be a System  $\mathcal{T}$  term such that  $\text{fv}(t) = \{x_1, \dots, x_n\}$ ,  $n \geq 0$ , and  $x_1:A_1, \dots, x_n:A_n \vdash_{\mathcal{T}} t : T$ . Its compilation into System  $\mathcal{L}$  is defined as:  $[x_1^{A_1}] \dots [x_n^{A_n}] \langle t \rangle$ ,<sup>1</sup> where we assume without loss of generality that the variables are processed in lexicographic order, and  $\langle \cdot \rangle$ ,  $[\cdot]$  are defined below by induction. Note that  $[x]t$  is only defined when  $x \in \text{fv}(t)$ .

$$\langle x \rangle = x$$

$$\langle su \rangle = \langle s \rangle \langle u \rangle$$

$$\begin{aligned} \langle \lambda x.u \rangle &= \lambda x.[x] \langle u \rangle, \text{ if } x \in \text{fv}(u) \\ &= \lambda x.\mathcal{E}(x, \langle A \rangle) \langle u \rangle, \text{ otherwise,} \end{aligned}$$

where  $x_1:A_1, \dots, x_n:A_n \vdash_{\mathcal{T}} t : A \rightarrow B = T$  by Lemma 1

$$\langle 0 \rangle = 0$$

$$\langle S u \rangle = S \langle u \rangle$$

$$\langle \text{iter } n \ u \ v \rangle = \text{iter } \langle n \rangle \ \langle u \rangle \ \langle v \rangle$$

$$\langle \langle s, u \rangle \rangle = \langle \langle s \rangle, \langle u \rangle \rangle$$

$$\langle \pi_1 s \rangle = \text{let } \langle x, y \rangle = \langle s \rangle \text{ in } \mathcal{E}(y, \langle B \rangle) x,$$

where  $x_1:A_1, \dots, x_n:A_n \vdash_{\mathcal{T}} t : A \times B = T$  by Lemma 1

$$\langle \pi_2 s \rangle = \text{let } \langle x, y \rangle = \langle s \rangle \text{ in } \mathcal{E}(x, \langle A \rangle) y,$$

where  $x_1:A_1, \dots, x_n:A_n \vdash_{\mathcal{T}} t : A \times B = T$  by Lemma 1

$$[x](S u) = S([x]u)$$

$$[x]x = x$$

$$[x](\lambda y.u) = \lambda y.[x]u$$

$$[x^A](su) = \begin{cases} C_{x:A}^{x_1, x_2}(A_{x_1}^x s)(A_{x_2}^x u) & x \in \text{fv}(s), x \in \text{fv}(u) \\ ([x]s)u & x \in \text{fv}(s), x \notin \text{fv}(u) \\ s([x]u) & x \in \text{fv}(u), x \notin \text{fv}(s) \end{cases}$$

$$[x^A]\langle s, u \rangle = \begin{cases} C_{x:A}^{x_1, x_2} \langle A_{x_1}^x s, A_{x_2}^x u \rangle, & x \in \text{fv}(s), x \in \text{fv}(u) \\ \langle [x]s, u \rangle, & x \in \text{fv}(s), x \notin \text{fv}(u) \\ \langle s, [x]u \rangle, & x \in \text{fv}(u), x \notin \text{fv}(s) \end{cases}$$

<sup>1</sup> We will omit the types of the variables  $x_1, \dots, x_n$  in the compilation when they are not necessary.

$$\begin{aligned}
[x^A](\text{let } \langle y, z \rangle = s \text{ in } u) &= \begin{cases} \text{let } \langle y, z \rangle = [x]s \text{ in } u & x \in \text{fv}(s), x \notin \text{fv}(u) \\ \text{let } \langle y, z \rangle = s \text{ in } [x]u & x \notin \text{fv}(s), x \in \text{fv}(u) \\ C_{x:A}^{x_1, x_2}(\text{let } \langle y, z \rangle = A_{x_1}^x s \text{ in } A_{x_2}^x u) & x \in \text{fv}(s), x \in \text{fv}(u) \end{cases} \\
[x^A](\text{iter } n \ u \ v) &= \begin{cases} \text{iter } [x]n \ u \ v & x \in \text{fv}(n), x \notin \text{fv}(uv) \\ \text{iter } n \ [x]u \ v & x \notin \text{fv}(nv), x \in \text{fv}(u) \\ \text{iter } n \ u \ [x]v & x \notin \text{fv}(nu), x \in \text{fv}(v) \\ C_{x:A}^{x_1, x_2} \text{iter } (A_{x_1}^x n) \ (A_{x_2}^x u) \ v & x \in \text{fv}(n) \cap \text{fv}(u), x \notin \text{fv}(v) \\ C_{x:A}^{x_1, x_3} \text{iter } (A_{x_1}^x n) \ u \ (A_{x_3}^x v) & x \in \text{fv}(n) \cap \text{fv}(v), x \notin \text{fv}(u) \\ C_{x:A}^{x_2, x_3} \text{iter } n \ (A_{x_2}^x u) \ (A_{x_3}^x v) & x \notin \text{fv}(n), x \in \text{fv}(u) \cap \text{fv}(v) \\ C_{x:A}^{x_1, x_2, x_3} \text{iter } (A_{x_1}^x n) \ (A_{x_2}^x u) \ (A_{x_3}^x v) & x \in \text{fv}(n) \cap \text{fv}(u) \cap \text{fv}(v) \end{cases}
\end{aligned}$$

where the variables  $x_1, x_2$  and  $x_3$  above are assumed fresh.

**Examples.** To illustrate the encoding, we show the compilation of the combinators:

- $\langle \lambda x.x \rangle = \lambda x.x$ .
- $\langle \lambda xyz.xz(yz) \rangle = \lambda xyz.\text{let } \langle z_1, z_2 \rangle = D_2^A z \text{ in } xz_1(yz_2)$ ,  
with  $\vdash_{\mathcal{T}} \lambda xyz.xz(yz) : (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$ .
- $\langle \lambda xy.x \rangle = \lambda xy.\mathcal{E}(y, B)x$ , with  $\vdash_{\mathcal{T}} \lambda xy.x : A \rightarrow B \rightarrow A$ .

A more interesting example is the following definition of the predecessor function. Define first the term:  $P = \lambda n.\text{iter } n \ \langle I, 0 \rangle \ \lambda w.\text{let } \langle u, v \rangle = w \text{ in } \langle \text{succ}, uv \rangle$ , where  $\text{succ} = \lambda x.S \ x : \mathbb{N} \multimap \mathbb{N}$ . Since the function in the iterator is closed, we have:

$$\begin{aligned}
P \ 0 &\longrightarrow \langle I, 0 \rangle \\
P \ (S \ n) &\longrightarrow^* \langle \text{succ}, n \rangle
\end{aligned}$$

To define the predecessor function we just need to add a projection. We use the encoding of  $\pi_2 t$ ,  $\langle \pi_2 t \rangle$ , given in Definition 28:

$$\text{Pred} = \lambda n.\text{let } P \ n = \langle x, y \rangle \text{ in } (\text{iter } (x \ 0) \ I \ I)y.$$

The rest of this section is devoted to the proof of correctness of the compilation function; for more details and examples see [2]. First we show that the result of the compilation of a System  $\mathcal{T}$  term satisfies the linearity constraints of System  $\mathcal{L}$ .

**Definition 29** Consider a finite set of variables  $X$ . We say that a term  $t$  is in  $\Lambda_{\mathcal{L}}^{+X}$ , if  $t$  is a term built using the syntax of  $\Lambda_{\mathcal{L}}$ , for which the linearity conditions hold for all its bound variables and also for the free variables in  $X$ . Note that, if  $X = \text{fv}(t)$ , then  $t$  is a term in  $\Lambda_{\mathcal{L}}$  (i.e., the variable constraints hold for all its variables).

**Example 30** Let  $X_1 = \{x, y, z\}$  and  $X_2 = \{x, y\}$ . The term  $S^3(0)$  is in both  $\Lambda_{\mathcal{L}}^{+X_1}$  and  $\Lambda_{\mathcal{L}}^{+X_2}$ . The term  $\langle xz, yz \rangle$  is in  $\Lambda_{\mathcal{L}}^{+X_2}$ , but not in  $\Lambda_{\mathcal{L}}^{+X_1}$ .

**Proposition 31** If  $t$  is a term in  $\Lambda_{\mathcal{L}}^{+X}$ , and  $x \notin \mathbf{fv}(t)$ , then  $t$  is in  $\Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$ .

**Lemma 32** If  $t$  is a term in  $\Lambda_{\mathcal{L}}^{+X}$ , and  $x$  is a free variable in  $t$ , then:

- (1)  $\mathbf{fv}([x]t) = \mathbf{fv}(t)$
- (2)  $[x]t$  is a term in  $\Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$

**PROOF.** First note that the duplicating term  $D$  is a closed term, in  $\Lambda_{\mathcal{L}}^{+X}$ , for any set of variables  $X$ . The lemma is proved by simultaneous induction on  $t$ . We show the case of an application term (pairs and iterators are treated similarly, the other cases follow directly by induction):

- $t \equiv uv$ , and  $x \in \mathbf{fv}(u)$ ,  $x \notin \mathbf{fv}(v)$  (the case  $x \notin \mathbf{fv}(u)$ ,  $x \in \mathbf{fv}(v)$  is similar).
  - (1)  $\mathbf{fv}([x]uv) = \mathbf{fv}([x]u) \cup \mathbf{fv}(v) \stackrel{(\text{I.H.})}{=} \mathbf{fv}(u) \cup \mathbf{fv}(v) = \mathbf{fv}(uv)$
  - (2)  $[x](uv) = ([x]u)v$ . By induction hypothesis  $[x]u$  is in  $\Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$  and  $v$  is in  $\Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$  by Proposition 31. By hypothesis  $\mathbf{fv}(u) \cap \mathbf{fv}(v) \cap X = \emptyset$ . By induction hypothesis (1)  $\mathbf{fv}([x]u) = \mathbf{fv}(u)$ , thus  $\mathbf{fv}([x]u) \cap \mathbf{fv}(v) \cap (X \cup \{x\}) = \emptyset$  (because  $x \notin \mathbf{fv}(v)$ ). Therefore  $([x]u)v$  is in  $\Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$ .
- $t \equiv uv$ , and  $x \in \mathbf{fv}(u)$ ,  $x \in \mathbf{fv}(v)$ . Then
  - (1) In the following, note that if  $x \in t$ , then  $\mathbf{fv}(t[y/x]) = (\mathbf{fv}(t) \setminus \{x\}) \cup \{y\}$ .

$$\begin{aligned}
 \mathbf{fv}([x]uv) &= \mathbf{fv}(\mathbf{let} \langle x_1, x_2 \rangle = Dx \mathbf{in} (([x]u)[x_1/x])([x]v)[x_2/x]) \\
 &= \mathbf{fv}(Dx) \cup (\mathbf{fv}(((x]u)[x_1/x])([x]v)[x_2/x])) \setminus \{x_1, x_2\} \\
 &= \{x\} \cup ((\mathbf{fv}([x]u) \setminus \{x\}) \cup \{x_1\} \cup ((\mathbf{fv}([x]v) \setminus \{x\}) \cup \{x_2\}) \setminus \{x_1, x_2\}) \\
 &\stackrel{(\text{I.H.})}{=} \{x\} \cup ((\mathbf{fv}(u) \setminus \{x\}) \cup (\mathbf{fv}(v) \setminus \{x\})) \\
 &= \mathbf{fv}(uv)
 \end{aligned}$$

- (2)  $[x](uv) = \mathbf{let} \langle x_1, x_2 \rangle = Dx \mathbf{in} (([x]u)[x_1/x])([x]v)[x_2/x]$   
By induction hypothesis (1):

$$\begin{aligned}
 \mathbf{fv}([x]u) &= \mathbf{fv}(u) = Y_u \cup \{x\} \\
 \mathbf{fv}([x]v) &= \mathbf{fv}(v) = Y_v \cup \{x\}
 \end{aligned}$$

By hypothesis,  $\mathbf{fv}(u) \cap \mathbf{fv}(v) \cap X = \emptyset$ , thus  $Y_u \cap Y_v \cap X = \emptyset$ . Note that

$$\begin{aligned}
 \mathbf{fv}([x]u)[x_1/x] &= Y_u \cup \{x_1\} \\
 \mathbf{fv}([x]v)[x_2/x] &= Y_v \cup \{x_2\}
 \end{aligned}$$

Therefore  $\text{fv}(A_{x_1}^x u) \cap \text{fv}(A_{x_2}^x v) \cap (X \cup \{x\}) = \emptyset$ , then  $(A_{x_1}^x u)(A_{x_2}^x v) \in \Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$ . Also,  $\text{fv}(Dx) = \{x\}$  and  $Dx$  is a term in  $\Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$  for any set of variables  $X$ . Thus,  $\text{fv}(Dx) \cap \text{fv}((A_{x_1}^x u)(A_{x_2}^x v)) \cap (X \cup \{x\}) = \emptyset$ , and  $x_1, x_2 \in \text{fv}(A_{x_1}^x u)(A_{x_2}^x v)$ , therefore  $\text{let } \langle x_1, x_2 \rangle = Dx \text{ in } (([x]u)[x_1/x])(([x]v)[x_2/x]) \in \Lambda_{\mathcal{L}}^{+(X \cup \{x\})}$ .

□

**Lemma 33** *Let  $t$  be a System  $\mathcal{T}$  term, then  $\text{fv}(\langle t \rangle) = \text{fv}(t)$ .*

**PROOF.** By a routine induction on  $t$ , using Lemma 32.

**Lemma 34** *Let  $t$  be a System  $\mathcal{T}$  term, then  $\langle t \rangle$  is a term in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ .*

**PROOF.** The only non trivial cases are abstractions and projections.

- $t \equiv \lambda x.u$ , and  $x \in \text{fv}(u)$ . Then  $\langle t \rangle = \lambda x.[x]\langle u \rangle$ . By induction hypothesis  $\langle u \rangle$  is a term in  $\Lambda_{\mathcal{L}}^{+\emptyset}$  and by Lemma 32  $x \in \text{fv}([x]\langle u \rangle)$ , therefore  $\lambda x.[x]\langle u \rangle$  is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ .
- $t \equiv \lambda x.u$ , and  $x \notin \text{fv}(u)$ . Then  $\langle t \rangle = \lambda x.(\mathcal{E}(x, \langle A \rangle)\langle u \rangle)$ . By induction hypothesis  $\langle u \rangle$  is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ , and since  $\mathcal{E}(x, \langle A \rangle)$  is a System  $\mathcal{L}$  term, therefore is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ . Since  $\text{fv}(\mathcal{E}(x, \langle A \rangle)) \cap \text{fv}(\langle u \rangle) = \emptyset$ , then  $\mathcal{E}(x, \langle A \rangle)\langle u \rangle$  is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ , and since  $x \in \text{fv}(\mathcal{E}(x, \langle A \rangle)\langle u \rangle)$ , then  $\lambda x.(\mathcal{E}(x, \langle A \rangle)\langle u \rangle)$  is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ .
- $t \equiv \pi_1(u)$ . Then  $\langle t \rangle = \text{let } \langle x, y \rangle = \langle u \rangle \text{ in } \mathcal{E}(y, \langle A_2 \rangle)x$ . By induction hypothesis  $\langle u \rangle$  is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ . The terms  $\mathcal{E}(y, \langle A_2 \rangle)$  and  $x$  are both in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ , and since  $\text{fv}(\mathcal{E}(y, \langle A_2 \rangle)) \cap \text{fv}(x) = \emptyset$ , then  $\mathcal{E}(y, \langle A_2 \rangle)x$  is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ , and  $x, y \in \text{fv}(\mathcal{E}(y, \langle A_2 \rangle)x)$ , therefore  $\text{let } \langle x, y \rangle = \langle u \rangle \text{ in } \mathcal{E}(y, \langle A_2 \rangle)x$  is in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ .

**Lemma 35** *If  $t$  is a System  $\mathcal{T}$  term, then:*

- (1)  $\text{fv}([x_1] \cdots [x_n]\langle t \rangle) = \text{fv}(t)$ .
- (2) *If  $\text{fv}(t) = \{x_1, \dots, x_n\}$ , then  $[x_1] \cdots [x_n]\langle t \rangle$  is in  $\Lambda_{\mathcal{L}}$ .*

**PROOF.**

- (1) By induction on the number of variables  $n$  using Lemmas 32 and 33.
- (2) By Lemma 34,  $\langle t \rangle$  is a term in  $\Lambda_{\mathcal{L}}^{+\emptyset}$ . By induction on the number of variables  $n$ , using part 2 of Lemma 32,  $[x_1] \cdots [x_n]\langle t \rangle$  is a term in  $\Lambda_{\mathcal{L}}^{+\text{fv}(t)}$ , therefore a term in  $\Lambda_{\mathcal{L}}$ . □

Lemma 35 states that the result of the compilation of a System  $\mathcal{T}$  term is a System  $\mathcal{L}$  term. We will now prove that the compilation of a typable term is

also typable (Theorem 38). For this, we will show that the terms obtained in the intermediate steps of the compilation can be typed in a hybrid system, obtained from System  $\mathcal{L}$  by allowing weakening and contraction only for variables in a certain set  $X$  (i.e., we relax the linear constraints for the variables in  $X$ ). Typing judgements in this system will be denoted  $\Gamma \vdash_{\mathcal{L}+X} t : T$ . A typing context  $\Gamma$  is still a set of assumptions of the form  $x:A$ , where each variable occurs at most once. The axiom for System  $\mathcal{L}+X$  is:

$$\Gamma, x : A \vdash x : A \quad \text{where } \text{dom}(\Gamma) \subseteq X$$

The typing rules are the same as in System  $\mathcal{L}$ , except that in the rules that split a typing context between the premisses in System  $\mathcal{L}$ , now we can share variables in  $X$ . For instance, in the rule  $\multimap$ Elim (see Figure 3) used to type application terms  $uv$ , we may have  $\text{fv}(u) \cap \text{fv}(v) \subseteq X$  and  $\text{dom}(\Gamma \cap \Delta) \subseteq X$ .

Note that if  $X = \emptyset$  then  $\mathcal{L}+X$  coincides with System  $\mathcal{L}$ .

We will denote by  $\Gamma|_X$  the restriction of  $\Gamma$  to the variables in  $X$ .

**Lemma 36** *If  $\Gamma \vdash_{\mathcal{L}+X} t : A$ , where  $\text{dom}(\Gamma) = \text{fv}(t)$  and  $x \in X \subseteq \text{fv}(t)$ , then  $\Gamma \vdash_{\mathcal{L}+X'} [x]t : A$ , where  $X' = X \setminus \{x\}$ .*

**PROOF.** By induction on  $t$ , using the fact that  $x : A \vdash_{\mathcal{L}+\emptyset} Dx : A \otimes A$ . We show the cases for variable and application.

- $t \equiv x$ . Then  $[x]x = x$ , and using the axiom we obtain both  $x : A \vdash_{\mathcal{L}+\{x\}} x : A$  and  $x : A \vdash_{\mathcal{L}+\emptyset} x : A$ .
- $t \equiv uv$ , and  $x \in \text{fv}(u)$ ,  $x \notin \text{fv}(v)$  (the case where  $x \notin \text{fv}(u)$ ,  $x \in \text{fv}(v)$  is similar). Then  $[x]uv = ([x]u)v$  and  $\Gamma \vdash_{\mathcal{L}+X} uv : A$ . Let  $\Gamma_1 = \Gamma|_{\text{fv}(u)}$  and  $\Gamma_2 = \Gamma|_{\text{fv}(v)}$ . Then  $\Gamma_1 \vdash_{\mathcal{L}+X} u : B \multimap A$  and  $\Gamma_2 \vdash_{\mathcal{L}+X} v : B$ , where  $\Gamma_1$  and  $\Gamma_2$  can only share variables in  $X$ . By induction hypothesis  $\Gamma_1 \vdash_{\mathcal{L}+X'} [x]u : B \multimap A$ . Also, since  $x \notin \text{fv}(v)$  and  $\text{dom}(\Gamma_2) = \text{fv}(v)$ , we have  $\Gamma_2 \vdash_{\mathcal{L}+X'} v : B$ . Therefore  $\Gamma \vdash_{\mathcal{L}+X'} ([x]u)v : A$ .
- $t \equiv uv$ ,  $x \in \text{fv}(u)$ , and  $x \in \text{fv}(v)$ . Let  $\Gamma_1 = \Gamma|_{\text{fv}(u) \setminus \{x\}}$  and  $\Gamma_2 = \Gamma|_{\text{fv}(v) \setminus \{x\}}$  and assume  $C$  is the type associated to  $x$  in  $\Gamma$ . Then  $\Gamma_1, x : C \vdash_{\mathcal{L}+X} u : B \multimap A$  and  $\Gamma_2, x : C \vdash_{\mathcal{L}+X} v : B$ . By induction hypothesis  $\Gamma_1, x : C \vdash_{\mathcal{L}+X'} [x]u : B \multimap A$ , and  $\Gamma_2, x : C \vdash_{\mathcal{L}+X'} [x]v : B$ . Thus  $\Gamma_1, x_1 : C \vdash_{\mathcal{L}+X'} ([x]u)[x_1/x] : B \multimap A$ , and  $\Gamma_2, x_2 : C \vdash_{\mathcal{L}+X'} ([x]v)[x_2/x] : B$ . Therefore  $\Gamma_1, x_1 : C, \Gamma_2, x_2 : C \vdash_{\mathcal{L}+X'} (A_{x_1}^x u)(A_{x_2}^x v) : A$ . Also  $x : C \vdash_{\mathcal{L}+\emptyset} Dx : C \otimes C$ , therefore  $\Gamma_1, \Gamma_2, x : C \vdash_{\mathcal{L}+X'} \text{let } \langle x_1, x_2 \rangle = Dx \text{ in } (A_{x_1}^x u)(A_{x_2}^x v) : A \quad \square$

**Lemma 37** *If  $\Gamma \vdash_{\mathcal{T}} t : A$ , then  $\langle \Gamma|_{\text{fv}(t)} \rangle \vdash_{\mathcal{L}+\text{fv}(t)} \langle t \rangle : \langle A \rangle$ .*

**PROOF.** By induction on the type derivation  $\Gamma \vdash_{\mathcal{T}} t : A$ . We distinguish cases depending on the last typing rule applied. Below we show a few interesting cases.

- $\rightarrow$ Intro:  $\Gamma \vdash_{\mathcal{T}} \lambda x.u : A_1 \rightarrow A_2$  if  $\Gamma, x : A_1 \vdash_{\mathcal{T}} u : A_2$ . By induction hypothesis,  $\langle \langle \Gamma, x : A_1 \rangle_{|\mathbf{fv}(u)} \rangle \vdash_{\mathcal{L}+\mathbf{fv}(u)} \langle u \rangle : \langle A_2 \rangle$ . There are two cases:
  - $x \in \mathbf{fv}(u)$ . Then  $\langle \lambda x.u \rangle = (\lambda x.[x]\langle u \rangle)$ , and  $(\Gamma, x:A_1)_{|\mathbf{fv}(u)} = \Gamma_{|\mathbf{fv}(u)}, x:A_1$ . Thus  $\langle \Gamma_{|\mathbf{fv}(u)}, x:\langle A_1 \rangle \rangle \vdash_{\mathcal{L}+\mathbf{fv}(u)} \langle u \rangle : \langle A_2 \rangle$ .  
By Lemma 36,  $\langle \Gamma_{|\mathbf{fv}(u)}, x:\langle A_1 \rangle \rangle \vdash_{\mathcal{L}+\mathbf{fv}(t)} [x]\langle u \rangle : \langle A_2 \rangle$ .  
Hence,  $\langle \Gamma_{|\mathbf{fv}(t)} \rangle \vdash_{\mathcal{L}+\mathbf{fv}(t)} \lambda x.[x]\langle u \rangle : \langle A_1 \rangle \multimap \langle A_2 \rangle$ .
  - $x \notin \mathbf{fv}(u)$ , thus  $\mathbf{fv}(t) = \mathbf{fv}(u)$ . Then  $\langle \lambda x.u \rangle = (\lambda x.\mathcal{E}(x, \langle A_1 \rangle)\langle u \rangle)$ ,  $(\Gamma, x : A_1)_{|\mathbf{fv}(u)} = \Gamma_{|\mathbf{fv}(t)}$  and  $\langle \Gamma_{|\mathbf{fv}(t)} \rangle \vdash_{\mathcal{L}+\mathbf{fv}(t)} \langle u \rangle : \langle A_2 \rangle$ . By Lemma 22,  $x:\langle A_1 \rangle \vdash_{\mathcal{L}^\emptyset} \mathcal{E}(x, \langle A_1 \rangle) : \langle A_2 \rangle \multimap \langle A_2 \rangle$ , hence  $\langle \Gamma_{|\mathbf{fv}(t)}, x:\langle A_1 \rangle \rangle \vdash_{\mathcal{L}+\mathbf{fv}(t)} \mathcal{E}(x, \langle A_1 \rangle)\langle u \rangle : \langle A_2 \rangle$ .  
Thus  $\langle \Gamma_{|\mathbf{fv}(t)} \rangle \vdash_{\mathcal{L}+\mathbf{fv}(t)} \lambda x.\mathcal{E}(x, \langle A_1 \rangle)\langle u \rangle : \langle A_1 \rangle \multimap \langle A_2 \rangle$ .
- $\times$ Elim:  $\Gamma \vdash_{\mathcal{T}} \pi_1 u : A_1$  if  $\Gamma \vdash_{\mathcal{T}} u : A_1 \times A_2$ . By induction hypothesis  $\langle \Gamma_{|\mathbf{fv}(u)} \rangle \vdash_{\mathcal{L}+\mathbf{fv}(u)} \langle u \rangle : \langle A_1 \rangle \otimes \langle A_2 \rangle$ . By Lemma 22  $y : \langle A_2 \rangle \vdash_{\mathcal{L}^\emptyset} \mathcal{E}(y, \langle A_2 \rangle) : \langle A_1 \rangle \multimap \langle A_1 \rangle$ . Also  $x : \langle A_1 \rangle \vdash_{\mathcal{L}^\emptyset} x : \langle A_1 \rangle$ , therefore  $y : \langle A_2 \rangle, x : \langle A_1 \rangle \vdash_{\mathcal{L}^\emptyset} \mathcal{E}(y, \langle A_2 \rangle)x : \langle A_1 \rangle$ .  
Thus  $\langle \Gamma_{|\mathbf{fv}(t)} \rangle \vdash_{\mathcal{L}+\mathbf{fv}(t)} \text{let } \langle x, y \rangle = \langle u \rangle \text{ in } \mathcal{E}(y, \langle A_2 \rangle)x : \langle A_1 \rangle$ . □

**Theorem 38** *If  $\Gamma \vdash_{\mathcal{T}} t : T$  and  $\mathbf{fv}(t) = \{x_1, \dots, x_n\}$  then*

$$\langle \Gamma_{|\mathbf{fv}(t)} \rangle \vdash_{\mathcal{L}} [x_1] \dots [x_n] \langle t \rangle : \langle T \rangle.$$

**PROOF.** By induction on the number of free variables of  $t$ , using Lemmas 36 and 37. □

We will now prove that we can simulate System  $\mathcal{T}$  evaluations. First we need a substitution lemma.

**Lemma 39** (1) *If  $t$  is in  $\Lambda_{\mathcal{L}}^{+X}$ ,  $x, y \in \mathbf{fv}(t)$ , and  $\mathbf{fv}(u) = \emptyset$ , then*

$$([y]t)[u/x] = [y](t[u/x]).$$

(2) *If  $t$  is a System  $\mathcal{T}$  term,  $x \in \mathbf{fv}(t)$ , and  $\mathbf{fv}(u) = \emptyset$ , then*

$$\langle t \rangle[\langle u \rangle/x] = \langle t[u/x] \rangle$$

**PROOF.** By induction on  $t$ , see Definition 29 and Lemma 32. □

**Lemma 40** *If  $t \in \Lambda_{\mathcal{L}}^{+X}$ ,  $x \in \mathbf{fv}(t)$ , and  $\mathbf{fv}(u) = \emptyset$  then  $([x]t)[u/x] \longrightarrow^* t[u/x]$ .*

**PROOF.** By induction on  $t$ ; below we show the case of application.

- $t \equiv sv$ ,  $x \in \text{fv}(s)$ ,  $x \notin \text{fv}(v)$  (the case  $x \notin \text{fv}(s)$ ,  $x \in \text{fv}(v)$  is similar).

$$\begin{aligned} ([x]sv)[u/x] &= (([x]s)v)[u/x] \\ &= (([x]s)[u/x])v \xrightarrow{(\text{I.H.})} (s[u/x])v = (sv)[u/x] \end{aligned}$$

- $t \equiv sv$ ,  $x \in \text{fv}(s), x \in \text{fv}(v)$ .

$$\begin{aligned} ([x]sv)[u/x] &= \text{let } \langle x_1, x_2 \rangle = Dx \text{ in } (([x]s)[x_1/x])(([x]v)[x_2/x])[u/x] \\ &= \text{let } \langle x_1, x_2 \rangle = Du \text{ in } (([x]s)[x_1/x])(([x]v)[x_2/x]) \\ &\longrightarrow^* \text{let } \langle x_1, x_2 \rangle = \langle u, u \rangle \text{ in } (([x]s)[x_1/x])(([x]v)[x_2/x]) \\ &\longrightarrow (([x]s)[u/x])(([x]v)[u/x]) \\ &\xrightarrow{(\text{I.H.})} (s[u/x])(v[u/x]) = (sv)[u/x] \quad \square \end{aligned}$$

**Theorem 41 (Simulation)** *Let  $t$  be a closed, typable System  $\mathcal{T}$  term, then:*

$$t \Downarrow u \Rightarrow \langle t \rangle \longrightarrow^* \langle u \rangle.$$

**PROOF.** By induction on  $t \Downarrow u$  (see Figure 2). We show two cases:

*Application.* By induction:  $\langle tu \rangle = \langle t \rangle \langle u \rangle \longrightarrow^* \langle \lambda x.t' \rangle \langle u \rangle$ . There are now two cases:

- If  $x \in \text{fv}(t')$  then using Lemma 40:

$$\langle \lambda x.t' \rangle \langle u \rangle = (\lambda x.[x]\langle t' \rangle) \langle u \rangle \longrightarrow ([x]\langle t' \rangle)[\langle u \rangle/x] \longrightarrow^* \langle t'[u/x] \rangle \longrightarrow^* \langle v \rangle$$

- Otherwise, using Lemmas 23 and 24:

$$\begin{aligned} \langle \lambda x.t' \rangle \langle u \rangle &= (\lambda x.\mathcal{E}(x, A)\langle t' \rangle) \langle u \rangle \\ &\longrightarrow^* (\mathcal{E}(\langle u \rangle, \langle A \rangle)\langle t' \rangle) \longrightarrow \langle t' \rangle = \langle t'[u/x] \rangle \longrightarrow^* \langle v \rangle \end{aligned}$$

*Projection.* By induction and Lemmas 23 and 24:

$$\begin{aligned} \langle \pi_1 t \rangle &= \text{let } \langle x, y \rangle = \langle t \rangle \text{ in } \mathcal{E}(y, \langle A \rangle)x \\ &\longrightarrow^* \text{let } \langle x, y \rangle = \langle \langle u, v \rangle \rangle \text{ in } \mathcal{E}(y, \langle A \rangle)x \\ &= \text{let } \langle x, y \rangle = \langle \langle u \rangle, \langle v \rangle \rangle \text{ in } \mathcal{E}(y, \langle A \rangle)x \longrightarrow \mathcal{E}(\langle v \rangle, \langle A \rangle)\langle u \rangle \longrightarrow^* \langle v \rangle \quad \square \end{aligned}$$

As a corollary we get that  $\mathcal{T} = \mathcal{L}$ .

## 7 Closed reduction, closed construction, and linearity

In [30] it was shown that the linear  $\lambda$ -calculus is the internal language for symmetric monoidal closed categories (the analogous result to the  $\lambda$ -calculus being the internal language to Cartesian Closed Categories). The addition of natural numbers and an iterator corresponds to adding a natural number object in the category. Note that, in this linear setting, the iterator is only allowed to iterate closed linear functions. More precisely, the typing rule for iterators requires the function to be typed in an empty environment, that is, iterators are “closed by *construction*”:

$$\frac{\Gamma \vdash n : \mathbf{N} \quad \Delta \vdash u : A \quad \vdash v : A \multimap A}{\Gamma, \Delta \vdash \text{iter } n \ u \ v}$$

Recall that in Section 3, we showed that a version of System  $\mathcal{T}$  with closed-at-construction iterators is as powerful as System  $\mathcal{T}$ . Although a closed-at-construction approach does not weaken System  $\mathcal{T}$ , the same does not hold in the presence of linearity. In order to show this, we will define two linear systems:  $\mathcal{L}^{\mathbf{N}}$  and  $\mathcal{L}_0^{\mathbf{N}}$ . System  $\mathcal{L}^{\mathbf{N}}$  has the same syntax as System  $\mathcal{L}$  and uses the closed reduction strategy, but its type system is more restrictive than System  $\mathcal{L}$ . System  $\mathcal{L}_0^{\mathbf{N}}$  does not restrict to closed reduction strategies, but, to be linear, it has to restrict the set of terms.

### 7.1 System $\mathcal{L}^{\mathbf{N}}$

System  $\mathcal{L}^{\mathbf{N}}$ 's syntax and reduction rules are the same as System  $\mathcal{L}$ 's; as a consequence, we inherit the following properties for the untyped calculus: Correctness of Substitution (Lemma 7), Correctness of  $\longrightarrow$  (Lemma 8), Confluence (Theorem 17).

$$\frac{\Gamma \vdash_{\mathcal{L}^{\mathbf{N}}} t : \mathbf{N} \quad \Delta \vdash_{\mathcal{L}^{\mathbf{N}}} u : \mathbf{N}}{\Gamma, \Delta \vdash_{\mathcal{L}^{\mathbf{N}}} \langle t, u \rangle : \mathbf{N} \otimes \mathbf{N}} \ (\otimes\text{Intro})$$

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

Fig. 5. Type System for System  $\mathcal{L}^{\mathbf{N}}$

We associate types to terms in System  $\mathcal{L}^{\mathbf{N}}$  using the same typing rules as for System  $\mathcal{L}$ , except for the ones involving pairs, which we replace by those given in Figure 5.

Subject Reduction for System  $\mathcal{L}^{\mathbf{N}}$  can be proved as for System  $\mathcal{L}$ .

Note that confluence of the untyped calculus, together with subject reduction, implies confluence of the typed calculus.

Since terms typable in System  $\mathcal{L}^N$  are also typable in System  $\mathcal{L}$ , we inherit the strong normalisation property.

## 7.2 System $\mathcal{L}_0^N$

The set of terms for System  $\mathcal{L}_0^N$  is built in the same way as for System  $\mathcal{L}^N$ , except that when building an iterator, we do not allow the iterated function to be an open term. Thus iterators in this system have the following definition (note the additional constraint  $\text{fv}(v) = \emptyset$ ):

$$\text{iter } t \ u \ v \quad \text{if } \text{fv}(t) \cap \text{fv}(u) = \emptyset \text{ and } \text{fv}(v) = \emptyset$$

We now define the reduction rules, and the typing rules, for System  $\mathcal{L}_0^N$ .

**Definition 42 (Reduction)** *Table 3 gives the reduction rules for System  $\mathcal{L}_0^N$ , substitution is a meta-operation defined as usual. Reductions can take place in any context.*

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

Table 3  
Reduction for System  $\mathcal{L}_0^N$

Correctness of Substitution is proved as for System  $\mathcal{L}$ , but  $\alpha$ -conversion must be used in substitution whenever necessary. Note that  $\alpha$ -conversion was not needed in System  $\mathcal{L}$  and therefore in System  $\mathcal{L}^N$ , because all the substitutions take a closed term.

**Lemma 43 (Correctness of substitution)** *Let  $t$  and  $u$  be terms in System  $\mathcal{L}_0^N$  and  $x \in \text{fv}(t)$ . Then  $t[u/x]$  is a term in System  $\mathcal{L}_0^N$ .*

**PROOF.** Straightforward induction on the structure of  $t$ .

**Lemma 44 (Correctness of  $\longrightarrow$ )** *Let  $t$  be a term in System  $\mathcal{L}_0^N$ , and  $t \longrightarrow u$ , then:*

- (1)  $\text{fv}(t) = \text{fv}(u)$ ;
- (2)  $u$  is a System  $\mathcal{L}_0^{\mathbb{N}}$  term.

**PROOF.** (Sketch) The only reduction rules that copy or erase terms, are the rules for iterators, which either copy or erase the iterated function. However, because of the condition that the iterated function must be closed when constructing the term  $\text{iter } t u v$ , then reducing an iterator will either copy or erase a closed term. Therefore the set of free variables is preserved and the term obtained is valid.

We associate types to terms in System  $\mathcal{L}_0^{\mathbb{N}}$  using the same typing rules as for System  $\mathcal{L}^{\mathbb{N}}$ , except for the (**Iter**) rule, where the context for the iterated function is always empty. Therefore, iterators in System  $\mathcal{L}_0^{\mathbb{N}}$  are typed in the following way:

$$\frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{N} \quad \Theta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : A \quad \vdash_{\mathcal{L}_0^{\mathbb{N}}} v : A \rightarrow A}{\Gamma, \Theta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \text{iter } t u v : A} \text{ (Iter)}$$

As before, Subject Reduction for System  $\mathcal{L}_0^{\mathbb{N}}$  can be proved as for System  $\mathcal{L}$ .

Note that any term typable in System  $\mathcal{L}_0^{\mathbb{N}}$  is also typable in System  $\mathcal{L}^{\mathbb{N}}$ , therefore in System  $\mathcal{L}$ . Thus, System  $\mathcal{L}_0^{\mathbb{N}}$  is strongly normalisable.

Confluence for typable terms in System  $\mathcal{L}_0^{\mathbb{N}}$  is a direct consequence of strong normalisation and the fact that the rules are non-overlapping (using Newman's Lemma [31]). Moreover, we can apply directly Klop's result [25] to the untyped calculus because the system is orthogonal (that is, left-linear and non-overlapping).

### 7.3 Primitive recursive functions and beyond

The encodings of the projections **fst** and **snd**, the copying function  $C$ , and the primitive recursive scheme, given for System  $\mathcal{L}$  in Section 5, satisfy the term conditions of System  $\mathcal{L}_0^{\mathbb{N}}$ , and therefore also those of System  $\mathcal{L}^{\mathbb{N}}$ . The reductions are valid in both systems, and the terms are typable in the more restricted type systems of System  $\mathcal{L}_0^{\mathbb{N}}$  and System  $\mathcal{L}^{\mathbb{N}}$ . Note that, to encode primitive recursive functions, one only needs pairs of natural numbers. Also, the encodings in Section 5 only iterate functions that are closed-by-construction, therefore typable in System  $\mathcal{L}_0^{\mathbb{N}}$ .

On the other hand, the encoding of Ackermann's function given for System  $\mathcal{L}$  using functions  $a$  and  $A$  is still valid in System  $\mathcal{L}^{\mathbb{N}}$ . However, note that

iter (S  $u$ ) (S 0)  $g$  cannot be typed in System  $\mathcal{L}_0^N$ , because  $g$  is a free variable. System  $\mathcal{L}^N$  allows building the term with the free variable  $g$ , but does not allow reduction until it is closed.

The fact that Ackermann's function cannot be defined in System  $\mathcal{L}_0^N$  is expected, as it can be seen as a subsystem of Dal Lago's linear language  $H(\emptyset)$  [11], albeit with a different syntax. Therefore System  $\mathcal{L}_0^N$  is strictly less powerful than System  $\mathcal{L}^N$  because we cannot define Ackermann's function in  $H(\emptyset)$  (see [11] for a proof of this result).

#### 7.4 Discussion

In Section 3, we showed that restricting the iterators using the closed-at-construction approach does not affect the computational power of System  $\mathcal{T}$ . A linear system with the same restriction is, however, strictly less powerful than a system using a closed-reduction approach, as shown in Section 7. A closed-reduction approach interacts well with linearity: System  $\mathcal{L}$  (a linear system with closed-reduction) proved to be as powerful as System  $\mathcal{T}$ .

It remains to understand the role of product types in the linear systems, or more precisely, the relationship between System  $\mathcal{L}$  and System  $\mathcal{L}^N$ . If one restricts pairs to natural numbers, then one loses the ability to define duplication of any term, as was done in Section 6. In [3], we presented a linear system, as powerful as System  $\mathcal{T}$ , where pairs were not crucial to define duplication, however, iterators were typed using a kind of polymorphic types, which we called iterator types. This suggests that pairs also play a role and add computational power to System  $\mathcal{L}$ .

## 8 Conclusions and future work

We have shown that in a linear  $\lambda$ -calculus with iterators (System  $\mathcal{L}$ ), the use of a 'closed-at-reduction' approach entails a gain in computational power, due to the fact that we can relax the constraints on the construction of iterator terms. Indeed, linear iterators with closed reduction have the computational power of System  $\mathcal{T}$ .

Several aspects of System  $\mathcal{L}$  remain to be studied:

- By the Curry-Howard isomorphism, the results can also be expressed as a property of the underlying logic (our translation from System  $\mathcal{T}$  to System  $\mathcal{L}$  eliminates Weakening and Contraction rules).

- Applications to category theory: Can this shed some new light on the relationship between Cartesian Closed Categories and Symmetric Monoidal Closed Categories, as outlined in the introduction?
- Does the technique extend to other typed  $\lambda$ -calculi, for instance the Calculus of Inductive Constructions [33]?
- System  $\mathcal{L}$  is not computationally complete (it is strongly normalising). A question that remains to study is whether it is possible to define a linear and computationally complete version of PCF using closed reductions.

**Acknowledgements.** This work was partially supported by the British Council Treaty of Windsor Programme, Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and FEDER/POSI, and Programa Gulbenkian de Estímulo à Investigação.

## 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 *Proceedings of CSL 2006, Computer Science Logic*, volume 4207 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
- [4] S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of closed-reduction strategies. In *Proceedings of WRS 2006, 6th International Workshop on Reduction Strategies in Rewriting and Programming, FLOC 2006, Seattle*, Electronic Notes in Theoretical Computer Science. Elsevier, 2007.
- [5] S. Alves and M. Florido. Weak linearization of the lambda calculus. *Theor. Comput. Sci.*, 342(1):79–103, 2005.
- [6] A. Asperti. Light affine logic. In *Proc. Logic in Computer Science (LICS'98)*. IEEE Computer Society, 1998.
- [7] A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM Transactions on Computational Logic*, 2002.
- [8] P. Baillot and V. Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS'04)*, LNCS. Springer Verlag, 2004.

- [9] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, second, revised edition, 1984.
- [10] N. Çağman and J. R. Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1–2):239–249, 1998.
- [11] U. Dal Lago. The geometry of linear higher-order recursion. In P. Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS'05*, pages 366–375. IEEE Computer Society Press, 2005.
- [12] M. Fernández and I. Mackie. Closed reduction in the  $\lambda$ -calculus. In J. Flum and M. Rodríguez-Artalejo, editors, *Proceedings of Computer Science Logic (CSL'99)*, volume 1683 of *Lecture Notes in Computer Science*, pages 220–234. Springer-Verlag, September 1999.
- [13] 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.
- [14] M. Gaboardi and L. Paolini. Syntactical, operational and denotational linearity. In *Workshop on Linear Logic, Ludics, Implicit Complexity and Operator Algebras. Dedicated to Jean-Yves Girard on his 60th birthday*, Siena, May 2007.
- [15] J. Girard. Light linear logic. *Information and Computation*, 1998.
- [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] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science*, 97:1–66, 1992.
- [21] J. Hindley. BCK-combinators and linear lambda-terms have types. *Theoretical Computer Science*, 64(1):97–105, 1989.
- [22] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proc. Logic in Computer Science (LICS'99)*. IEEE Computer Society, 1999.

- [23] S. Holmström. Linear functional programming. In T. Johnsson, S. L. Peyton Jones, and K. Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 13–32, 1988.
- [24] J. W. Klop. New fixpoint combinators from old. *Reflections on Type Theory*, 2007.
- [25] J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
- [26] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 2004.
- [27] 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.
- [28] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics Vol. 7. Cambridge University Press, 1986.
- [29] I. Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.
- [30] I. Mackie, L. Román, and S. Abramsky. An internal language for autonomous categories. *Journal of Applied Categorical Structures*, 1(3):311–343, 1993.
- [31] M. Newman. On theories with a combinatorial definition of “equivalence”. *Annals of Mathematics*, 43(2):223–243, 1942.
- [32] L. Paolini and M. Piccolo. Semantically linear programming languages. In *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 97–107, Valencia, Spain, July 2008. ACM.
- [33] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, 1993.
- [34] K. Terui. Affine lambda-calculus and polytime strong normalization. In *Proc. Logic in Computer Science (LICS’01)*. IEEE Computer Society Press, 2001.