The Power of Linear Functions*

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

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

Abstract. The linear lambda calculus is very weak in terms of expressive power: in particular, all functions terminate in linear time. In this paper we consider a simple extension with Booleans, natural numbers and a linear iterator. We show properties of this linear version of Gödel's System $\mathcal{T}$ and study the class of functions that can be represented. Surprisingly, this linear calculus is extremely expressive: it is as powerful as System $\mathcal{T}$.

1 Introduction

One of the many strands of work stemming from Girard's Linear Logic [8] is the area of linear functional programming (see for instance [1, 19, 14]). These languages are based on a version of the λ-calculus with a type system corresponding to intuitionistic linear logic. One of the features of the calculus (which can be seen as a minimal functional programming language) is that it provides 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) λ-calculus, and is summarised by just the following three rules:

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B} \qquad \frac{\Gamma \vdash t : A \multimap B \qquad \Delta \vdash u : A}{\Gamma, \Delta \vdash tu : B}$$

Due to the typing constraints—there is no contraction or weakening rule—terms are linear. Reduction is given by the usual β-reduction rule, but since there is no duplication or erasing of terms during reduction, this calculus has limited computational power—all functions terminate in linear time [12].

* Research partially supported by the Treaty of Windsor Grant: "Linearity: Programming Languages and Implementations", and by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, FCT* and *FEDER/POSI*.

** Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

Our work builds this language up by introducing: pairs, Booleans with a conditional, and natural numbers with the corresponding iterator, to obtain a *linear* version of Gödel's System $\mathcal{T}$ which we call System $\mathcal{L}$. The study of System $\mathcal{L}$ led us to the discovery of an interesting interplay between linearity and iteration in Gödel's original System $\mathcal{T}$. We will show that there is a great deal of redundancy in Gödel's System $\mathcal{T}$ and the same computational power can be achieved in a much more restricted system. Gödel's System $\mathcal{T}$ is a formalism built from the simply typed λ-calculus, adding numbers and Booleans, and a recursion operator. It is a very simple system, yet has 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 which has these two ingredients: System $\mathcal{L}$.

It is interesting to note that, in contrast with previous linear versions of System $\mathcal{T}$ (e.g., [16, 13]), System $\mathcal{L}$ accepts iterators on *open* linear functions, since these terms are linear. Reduction is only performed on those terms if the function becomes closed (i.e., reduction does not create non-linear terms). This design choice has an enormous impact in the computation power of the calculus: we show that our calculus is as powerful as System $\mathcal{T}$, whereas previous linear calculi were strictly weaker (see [16]).

From another perspective there have been a number of calculi, again many based on linear logic, for capturing specific complexity classes ([2, 7, 11, 3, 15, 24, 4]). One of the main examples is that of *bounded linear logic* [11], which has as one of its main aims to find a calculus in-between the linear λ-calculus and that with the exponentials (specifically the polynomial time computable functions). There is also previous work that uses linear types to characterise computations with time bounds [13]. 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.

Summarising, this paper studies the computational power of a linear System $\mathcal{T}$, exposing the structure of the components of Gödel's System $\mathcal{T}$. We show that System $\mathcal{T}$ is intrinsically redundant, in that it has several ways of expressing duplication and erasure. Can one eliminate this redundancy? The answer is yes; in this paper we:

- define a linear λ-calculus with natural numbers and an iterator, and introduce iterative types and the closed reduction strategy for this calculus;
- show that we can define the whole class of primitive recursive functions in this calculus, and more general functions such as Ackermann's;
- demonstrate that this linear System $\mathcal{T}$ has the same computational power as the full System $\mathcal{T}$.

In the next section we recall the background material. In Section 3 we define System $\mathcal{L}$ and in Section 4 we demonstrate that we can encode the primitive recursive functions in this calculus, and even go considerably beyond this class of functions. In Section 5 we show how to encode Gödel's System $\mathcal{T}$. Finally we conclude the paper in Section 6.

2 Background

We assume the reader is familiar with the λ-calculus [5], and with the main notions on primitive recursive functions [23]. In this section we recall some notions on Gödel's System $\mathcal{T}$, for more details we refer to [10].

System $\mathcal{T}$ is the simply typed λ-calculus (with arrow types and products, and the usual β-reduction and projection rules) where two basic types have been added: numbers (built from 0 and S; we write $\bar{n}$ or $\mathsf{S}^n\ 0$ for $\underbrace{\mathsf{S}\ \ldots\ (\mathsf{S}}_{n}\ 0)$) and Booleans with a recursor and a conditional defined by the reduction rules:

$$\begin{array}{ll} \mathsf{R}\ 0\ u\ v \quad \longrightarrow u & \mathtt{cond}\ \mathsf{true}\ u\ v \longrightarrow u \\ \mathsf{R}\ (\mathsf{S}t)\ u\ v \longrightarrow v\ (\mathsf{R}\ t\ u\ v)\ t & \mathtt{cond}\ \mathsf{false}\ u\ v \longrightarrow v \end{array}$$

System $\mathcal{T}$ is confluent, strongly normalising and reduction preserves types (see [10] for the complete system and results). It is well-known that an iterator has the same computational power as the recursor. We will replace the recursor by a simpler iterator:

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

with the following typing rule:

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

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). We recall the following property, which is used in Section 5:

Lemma 1. – *If $\Gamma \vdash_{\mathcal{T}} \lambda x.u : T$ then $T = A \to B$ and $\Gamma, x : A \vdash_{\mathcal{T}} u : B$ for some A, B.*
– *If $\Gamma \vdash_{\mathcal{T}} \pi_1(s) : T$ then $\Gamma \vdash_{\mathcal{T}} s : T \times B$ for some B.*
– *If $\Gamma \vdash_{\mathcal{T}} \pi_2(s) : T$ then $\Gamma \vdash_{\mathcal{T}} s : A \times T$ for some A.*

We now 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.

$$\frac{v \text{ is a value}}{v \Downarrow v} \qquad \frac{t \Downarrow \lambda x.t' \quad t'[u/x] \Downarrow v}{tu \Downarrow v} \qquad \frac{t \Downarrow \langle s, s' \rangle \quad s \Downarrow v}{\pi_1(t) \Downarrow v} \qquad \frac{t \Downarrow \langle s, s' \rangle \quad s' \Downarrow v}{\pi_2(t) \Downarrow v}$$

$$\frac{t \Downarrow v}{\mathsf{S}\ t \Downarrow \mathsf{S}\ v} \qquad \frac{t \Downarrow \mathsf{S}^n\ 0 \qquad s^n(u) \Downarrow v}{\mathsf{iter}\ t\ u\ s \Downarrow v} \qquad \frac{b \Downarrow \mathsf{true} \qquad t \Downarrow v}{\mathtt{cond}\ b\ t\ u \Downarrow v} \qquad \frac{b \Downarrow \mathsf{false} \qquad u \Downarrow v}{\mathtt{cond}\ b\ t\ u \Downarrow v}$$

Values are terms of the form: $\mathsf{S}^n 0$, true, false, $\langle s, s' \rangle$, $\lambda x.s$.

Lemma 2 (Adequacy of $\cdot \Downarrow \cdot$ for System $\mathcal{T}$). *1. If $t \Downarrow v$ then $t \longrightarrow^* v$.*
2. If $\Gamma \vdash t : T$, t closed, then:

$$\begin{array}{ll} T = \mathsf{Nat} \ \Rightarrow t \Downarrow \mathsf{S}(\mathsf{S} \ldots (\mathsf{S}\ 0)) & T = A \times B \ \Rightarrow t \Downarrow \langle u, s \rangle \\ T = \mathsf{Bool} \Rightarrow t \Downarrow \mathsf{true} \text{ or } t \Downarrow \mathsf{false} & T = A \to B \Rightarrow\ t \Downarrow \lambda x.s \end{array}$$

A program in System $\mathcal{T}$ is a closed term at base type (Nat or Bool).

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

In this section we extend the linear λ-calculus [1] with numbers, Booleans, pairs, and an iterator, and we specify a reduction strategy inspired by closed reduction [6, 9]. We call this system System $\mathcal{L}$. We begin by defining the set of linear λ-terms, which are terms from the λ-calculus restricted in the following way ($\mathsf{fv}(t)$ denotes the set of free variables of t).

$$\begin{array}{ll} x & \\ \lambda x.t & \text{if } x \in \mathsf{fv}(t) \\ tu & \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \end{array}$$

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). Next we add to this linear λ-calculus: pairs, Booleans and numbers. Table 1 summarises the syntax of System $\mathcal{L}$.

Pairs:

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

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

Booleans: true and false, and a conditional:

$$\texttt{cond } t\ u\ v \qquad \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \text{ and } \mathsf{fv}(u) = \mathsf{fv}(v)$$

Note that this linear conditional uses the same resources in each branch.

Numbers: 0 and S, and an iterator:

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

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

Table 1.

Definition 1 (Closed Reduction). *Table 2 gives the reduction rules for System $\mathcal{L}$, substitution is a meta-operation defined as usual. Reductions can take place in any context.*

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

Table 2. Closed reduction

Reduction is weak: for example, $\lambda x.(\lambda y.y)x$ is a normal form. Note that all the substitutions created during reduction (rules *Beta, Let*) are closed; this corresponds to a closed-argument reduction strategy (called ca in [6]). Also note that *Iter* rules are only triggered when the function v is closed.

The following results are proved by induction, by showing that substitution and reduction preserve the variable constraints given in Table 1.

Lemma 3 (Correctness of Substitution). *Let t and u be valid terms, $x \in \mathsf{fv}(t)$, and $\mathsf{fv}(u) = \varnothing$, then $t[u/x]$ is valid.*

Lemma 4 (Correctness of $\longrightarrow$). *Let t be a valid term, and $t \longrightarrow u$, then:*

1. $\mathsf{fv}(t) = \mathsf{fv}(u)$*;*
2. *u is a valid term.*

Although reduction preserves the free variables of the term, a subterm of the form $\mathsf{iter}\ n\ u\ v$ may become closed after a reduction in a superterm, triggering in this way a reduction with an *Iter* rule.

3.1 Typed Terms

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

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

Definition 2. *Let $A_0, \ldots, A_n$ be a list of linear types (note that $A_0, \ldots, A_n$ cannot be empty). $It(A_0, \ldots, A_n)$ denotes a non-empty set of* iterative types *defined by induction on n:*

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

Axiom and **Structural Rule**:

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

Logical Rules:

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

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

Numbers

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

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

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

Booleans

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

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

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

Iterative types will serve to type the functions used in iterators. Note that $It(A_0) = It(A_0, A_0) = It(A_0, \ldots, A_0)$. We associate types to terms in System $\mathcal{L}$ using the typing rules given in Figure 1, where we use the following abbreviations: $\Gamma \vdash_{\mathcal{L}} t : It(A_0, \ldots, A_n)$ iff $\Gamma \vdash_{\mathcal{L}} t : B$ for each $B \in It(A_0, \ldots, A_n)$. We use a Curry-style type system; the typing rules specify how to assign types to untyped terms (there are no type decorations).

Note that the only structural rule in Figure 1 is Exchange, 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 $\mathsf{iter}\ t\ u\ v$, we check that t is a term of type Nat and that v and u are compatible. There are two cases: if t is $\mathsf{S}^n\ 0$ then we require v to be a function that can be iterated n times on u. Otherwise, if t is not (yet) a number, we require v to have a type that allows it to be iterated any number of times (i.e. u has type A and $v : A \multimap A$, for some type A). The typing of iterators is therefore more flexible than in System $\mathcal{T}$, but we will see that this extra flexibility does not compromise the confluence and strong normalisation of the system. Also note that we allow the typing of $\mathsf{iter}\ t\ u\ v$ even if v is open (in contrast with [16, 13]), but we do not allow reduction until v is closed. This

feature gives our system the full power of System $\mathcal{T}$ (whereas systems that do not allow building iter with v open are strictly weaker [16]).

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

Lemma 5. *If $\Gamma \vdash_{\mathcal{L}} t : A$ then $dom(\Gamma) = \mathsf{fv}(t)$.*

Theorem 1 (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 a Substitution Lemma: If $\Gamma, x : A \vdash_{\mathcal{L}} t : B$ and $\Delta \vdash_{\mathcal{L}} u : A$ (where $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$) then $\Gamma, \Delta \vdash_{\mathcal{L}} t[u/x] : B$. □

3.2 Strong Normalisation.

In System $\mathcal{L}$, every sequence of reductions starting from a typable term is finite (i.e. typable terms are strongly normalisable). Note that, although System $\mathcal{L}$ extends the linear λ-calculus (where every term is strongly normalisable), untyped terms of System $\mathcal{L}$ may have infinite reductions. Strong normalisation for System $\mathcal{L}$ is a consequence of strong normalisation for System $\mathcal{T}$. We start by defining a translation from System $\mathcal{L}$ into System $\mathcal{T}$.

Definition 3. *We define the compilation of types and terms in System $\mathcal{L}$ into System $\mathcal{T}$, denoted $[\![\cdot]\!]$, in the following way:*

$$\begin{array}{ll} [\![\mathsf{Nat}]\!] = \mathsf{Nat} & [\![\mathsf{Bool}]\!] = \mathsf{Bool} \\ [\![A \multimap B]\!] = [\![A]\!] \to [\![B]\!] & [\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!] \end{array}$$

$$\begin{array}{ll} [\![0]\!] & = 0 \\ [\![\mathsf{true}]\!] & = \mathsf{true} \\ [\![\mathsf{false}]\!] & = \mathsf{false} \\ [\![\mathsf{S}\ t]\!] & = S([\![t]\!]) \\ [\![x]\!] & = x \\ [\![\lambda y.t]\!] & = \lambda y.[\![t]\!] \\ [\![tu]\!] & = [\![t]\!][\![u]\!] \\ [\![\langle t, u\rangle]\!] & = \langle [\![t]\!], [\![u]\!]\rangle \\ [\![\mathtt{let}\ \langle x, y\rangle = t\ \mathtt{in}\ u]\!] & = [\![u]\!][(\pi_1[\![t]\!])/x][(\pi_2[\![t]\!])/y] \\ [\![\mathsf{cond}\ t\ u\ v]\!] & = \mathsf{cond}\ [\![t]\!]\ [\![u]\!]\ [\![v]\!] \\ [\![\mathsf{iter}\ t\ u\ v]\!] & = \begin{cases} [\![v]\!]^m([\![u]\!]) & \text{if } t = \mathsf{S}^m 0,\ m > 0 \\ \mathsf{iter}\ [\![t]\!]\ [\![u]\!]\ [\![v]\!] & \text{otherwise} \end{cases} \end{array}$$

If $\Gamma = x_1 : A_1, \ldots x_n : A_n$, then $[\![\Gamma]\!] = x_1 : [\![A_1]\!], \ldots x_n : [\![A_n]\!]$. Note that the translation of an iterator where the number of times to iterate is known and positive, develops this iteration. If it is zero or not known we use System $\mathcal{T}$'s iterator.

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

Proof (Sketch). Strong normalisation for System $\mathcal{L}$ is proved by mapping all the reduction steps in System $\mathcal{L}$ into one or more reduction steps in System $\mathcal{T}$. Notice that reduction steps of the form $\mathsf{iter}\ S^{m+1}0\ u\ v \longrightarrow v(\mathsf{iter}\ S^m 0\ u\ v)$ map into zero reduction steps in System $\mathcal{T}$, but we can prove that any sequence of reduction steps of that form is always terminating. □

3.3 Church-Rosser

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 Newmann's Lemma [21]). In fact, all System $\mathcal{L}$ terms are confluent even if they are non-terminating: this can be proved using parallel-reductions.

Theorem 3 (Church-Rosser). *If $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$, then there is a term t_3 such that $t_1 \longrightarrow^* t_3$ and $t_2 \longrightarrow^* t_3$.*

Theorem 4 (Adequacy). *If t is closed and typable, then one of the following holds:*

- $\vdash_{\mathcal{L}} t : \mathsf{Nat}$ *and* $t \longrightarrow^* \overline{n}$
- $\vdash_{\mathcal{L}} t : \mathsf{Bool}$ *and* $t \longrightarrow^* \mathsf{true}$ *or* $t \longrightarrow^* \mathsf{false}$
- $\vdash_{\mathcal{L}} t : A \multimap B$ *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 5, typing judgements for t have the form $\vdash_{\mathcal{L}} t : T$, and T is either Nat, Bool, $A \multimap B$ or $A \otimes B$. By Subject Reduction, Strong Normalisation, and Lemma 4, we know that t has a closed, typable normal form u. We show the case when $\vdash_{\mathcal{L}} u : \mathsf{Nat}$, the others follow with similar reasoning. Since u is a closed term of type Nat, it cannot be a variable, an abstraction or a pair. Hence u is either an application, a pair projection, a conditional, an iterator or a number.

- Let $u = u_1 u_2 \ldots u_n$, $n \geq 2$, such that u_1 is not an application. Then u_1 is closed, and since u is typable, u_1 must have an arrow type. But then by induction u_1 is an abstraction, and then the *Beta* rule would apply, contradicting our assumptions.
- Let $u = \mathtt{let}\ \langle x, y\rangle = s\ \mathtt{in}\ v$. Then s is closed and $\mathsf{fv}(v) = \{x, y\}$. Since u is typable, s has type $A \otimes B$, and by induction it should be a (closed) pair $\langle s_1, s_2\rangle$. But then the *Let* rule would apply contradicting our assumptions.
- Let $u = \mathsf{cond}\ n\ s\ v$. Then n, t, v are closed. Since u is typable, n must have a Boolean type, and by induction it should be either true or false. But then the *Cond* rule would apply contradicting our assumptions.
- Let $u = \mathsf{iter}\ n\ s\ v$. Since u is closed, so are n, t and v. Since u is typable n must be a term of type Nat, and by induction, n is a number. But then the *Iter* rule would apply contradicting our assumptions.

Thus, if $\vdash_{\mathcal{L}} t : \mathsf{Nat}$ then t reduces to a number. □

4 Primitive Recursive Functions Linearly

In this section we show how to define the primitive recursive functions in System $\mathcal{L}$. We conclude the section indicating that we can encode substantially more than primitive recursive functions.

Erasing linearly. Although System $\mathcal{L}$ is a linear calculus, we can erase numbers. In particular, we can define the projection functions $\mathsf{fst}, \mathsf{snd} : \mathsf{Nat} \otimes \mathsf{Nat} \multimap \mathsf{Nat}$:

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

Lemma 6. *For any numbers $\bar{a}$ and $\bar{b}$, $\mathsf{fst}\langle \bar{a}, \bar{b}\rangle \longrightarrow^* \bar{a}$ and $\mathsf{snd}\langle \bar{a}, \bar{b}\rangle \longrightarrow^* \bar{b}$.*

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

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

□

Copying linearly. We can also copy natural numbers in this linear calculus. For this, we define a function $C : \mathsf{Nat} \multimap \mathsf{Nat} \otimes \mathsf{Nat}$ that given a number $\bar{n}$ returns a pair $\langle \bar{n}, \bar{n}\rangle$: $C = \lambda x.\mathsf{iter}\ x\ \langle 0, 0\rangle\ (\lambda x.\mathtt{let}\ \langle a, b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle)$.

Lemma 7. *For any number $\bar{n}$, $C\ \bar{n} \longrightarrow^* \langle \bar{n}, \bar{n}\rangle$.*

Proof. By induction on $\bar{n}$.

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

□

It is easy to apply this technique to other data structures (e.g. linear lists). Note that we do not need iterative types for this (the standard typing of iterators is sufficient). More interestingly, we will show in Section 5 that iterators will indeed allow us to erase any closed term, and moreover copy any closed term.

Primitive Recursive Functions. 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 (see Section 2) using iter. First, assume h is defined by the following, simpler scheme (it uses n only once in the second equation):

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

Given a function $g : \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}$, let g' be the term:

$$\lambda y.\lambda z.\mathtt{let}\ \langle z_1, z_2\rangle = C\ z\ \mathtt{in}\ gz_1(yz_2) : (\mathsf{Nat} \multimap \mathsf{Nat}) \multimap (\mathsf{Nat} \multimap \mathsf{Nat})$$

then $h(x,n)$ is defined by the term $(\mathsf{iter}\ n\ f\ g')x : \mathsf{Nat}$, with $f : \mathsf{Nat} \multimap \mathsf{Nat}$. Indeed, we can show by induction that $(\mathsf{iter}\ n\ f\ g')x$, where x and n are numbers, reduces to the number $h(x,n)$; we use Lemma 7 to copy numbers:

$$\begin{array}{lll}
(\mathsf{iter}\ 0\ f\ g')x & \longrightarrow & (f\ x) = h(x,0) \\
(\mathsf{iter}\ (\mathsf{S}^{n+1}\ 0)\ f\ g')\ x & \longrightarrow^* & (\mathtt{let}\ \langle z_1, z_2\rangle = C\ z\ \mathtt{in}\ gz_1(yz_2)) \\
 & & [(\mathsf{iter}\ (\mathsf{S}^n\ 0)\ f\ g')/y, x/z] \\
 & \longrightarrow^* & \mathtt{let}\ \langle z_1, z_2\rangle = \langle x, x\rangle\ \mathtt{in}\ gz_1((\mathsf{iter}\ (\mathsf{S}^n\ 0)\ f\ g')z_2) \\
 & \longrightarrow & g\ x((\mathsf{iter}\ (\mathsf{S}^n\ 0)\ f\ g')x) = h(x, n+1) \text{ by induction.}
\end{array}$$

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: $h(x,n) = \mathtt{let}\ \langle n_1, n_2\rangle = C\ n\ \mathtt{in}\ sx$, where $s = \mathsf{iter}\ n_2\ f\ (\lambda y.\lambda z.\mathtt{let}\ \langle z_1, z_2\rangle = C\ z\ \mathtt{in}\ gz_1(yz_2)n_1)$. Note that the iterator in the encoding of $h(x,n)$ uses an open function, but it will be closed before reduction.

Beyond Primitive Recursion. Ackermann's function is a standard example of a non primitive recursive function:

$$\begin{array}{ll}
ack(0,n) & = \mathsf{S}\ n \\
ack(\mathsf{S}\ n, 0) & = ack(n, \mathsf{S}\ 0) \\
ack(\mathsf{S}\ n, \mathsf{S}\ m) & = ack(n, ack(\mathsf{S}\ n, m))
\end{array}$$

In a higher-order functional language, we have an alternative definition. Let $\mathsf{succ} = \lambda x.\mathsf{S}\ x : \mathsf{Nat} \multimap \mathsf{Nat}$, then $ack(m,n) = a\ m\ n$ where a is the function defined by:

$$\begin{array}{llll}
a\ 0 & = \mathsf{succ} & A\ g\ 0 & = g(\mathsf{S}\ 0) \\
a\ (\mathsf{S}\ n) & = A\ (a\ n) & A\ g\ (\mathsf{S}\ n) & = g(A\ g\ n)
\end{array}$$

Lemma 8. *Both definitions are equivalent: For all $x, y : \mathsf{Nat}$, $a\ x\ y = ack(x,y)$.*

Proof. By induction on x, proving first by induction on n that if $g = \lambda y.ack(x,y)$ then $A\ g\ n = ack(\mathsf{S}\ x, n)$. □

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

$$a\ n = \mathsf{iter}\ n\ \mathsf{succ}\ A : \mathsf{Nat} \multimap \mathsf{Nat} \qquad A\ g\ n = \mathsf{iter}\ (\mathsf{S}\ n)\ (\mathsf{S}\ 0)\ g : \mathsf{Nat}$$

We show by induction that this encoding is correct:

- $a\ 0 = \mathsf{iter}\ 0\ \mathsf{succ}\ A = \mathsf{succ}$
  $A\ g\ 0 = \mathsf{iter}\ (\mathsf{S}\ 0)\ (\mathsf{S}\ 0)\ g = g(\mathsf{S}\ 0)$
- $a\ (\mathsf{S}\ n) = \mathsf{iter}\ (\mathsf{S}^n\ 0)\ \mathsf{succ}\ A = A(\mathsf{iter}\ n\ \mathsf{succ}\ A) = A(a\ n)$
  $A\ g\ (\mathsf{S}\ n) = \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\ n))\ (\mathsf{S}\ 0)\ g = g(\mathsf{iter}\ (\mathsf{S}\ n)\ (\mathsf{S}\ 0)\ g) = g(A\ g\ n)$.

Then Ackermann's function can be defined in System $\mathcal{L}$ as:

$$ack(m,n) = (\mathsf{iter}\ m\ \mathsf{succ}\ (\lambda gu.\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g))\ n : \mathsf{Nat}$$

The correctness of this encoding follows directly from the lemma above. Note that $\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g$ cannot be typed in [16], 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.

5 The Power of System $\mathcal{L}$: System $\mathcal{T}$ Linearly

In this section we show how to compile System $\mathcal{T}$ programs into System $\mathcal{L}$, i.e. we show that System $\mathcal{T}$ and System $\mathcal{L}$ have the same computational power.

Explicit Erasing. In the linear λ-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 known as Solvability (see [5] for instance). Our goal in this section is to give an analogous result that allows us to obtain a general form of erasing.

Definition 4 (Erasing). *We define the following mutually recursive operations 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 $I = \lambda x.x$):*

$$\begin{array}{ll} \mathcal{E}(t, \mathsf{Nat}) = \mathsf{iter}\ t\ I\ I & \mathcal{E}(t, A \otimes B) = \mathtt{let}\ \langle x, y\rangle = t\ \mathtt{in}\ \mathcal{E}(x, A)\mathcal{E}(y, B) \\ \mathcal{E}(t, \mathsf{Bool}) = \mathtt{cond}\ t\ I\ I & \mathcal{E}(t, A \multimap B) = \mathcal{E}(t\mathcal{M}(A), B) \\ \mathcal{M}(\mathsf{Nat}) = 0 & \mathcal{M}(A \otimes B) = \langle \mathcal{M}(A), \mathcal{M}(B)\rangle \\ \mathcal{M}(\mathsf{Bool}) = \mathsf{true} & \mathcal{M}(A \multimap B) = \lambda x.\mathcal{E}(x, A)\mathcal{M}(B) \end{array}$$

Lemma 9. *If $\Gamma \vdash_{\mathcal{L}} t : T$ then:*

1. $\mathsf{fv}(\mathcal{E}(t, T)) = \mathsf{fv}(t)$ *and* $\Gamma \vdash_{\mathcal{L}} \mathcal{E}(t, T) : A \multimap A$.
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. We show two cases:

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

Lemma 10. *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 $\vdash_{\mathcal{L}} t[v/x] : T$. □

Lemma 11 (Erasing Lemma). *If $\vdash_{\mathcal{L}} t : T$ (i.e. t closed) then $\mathcal{E}(t, T) \longrightarrow^* I$.*

Proof. By induction on T, using Theorem 4:

$$\mathcal{E}(t, \mathsf{Nat}) = \mathsf{iter}\ t\ I\ I \longrightarrow^* \mathsf{iter}\ (\mathsf{S}^n 0)\ I\ I \longrightarrow^* I$$

$\mathcal{E}(t, \mathsf{Bool}) = \mathtt{cond}\ t\ I\ I \longrightarrow^* I$.

If $T = A \otimes B$, then $t \longrightarrow^* \langle a, b\rangle$ and by Theorem 1 and Lemma 4: $\vdash_{\mathcal{L}} a : A$ and $\vdash_{\mathcal{L}} b : B$. By induction, $\mathcal{E}(a, A) \longrightarrow^* I$ and $\mathcal{E}(b, B) \longrightarrow^* I$, therefore $\mathtt{let}\ \langle x, y\rangle = \langle a, b\rangle\ \mathtt{in}\ \mathcal{E}(x, A)\mathcal{E}(y, B) \longrightarrow^* I$.

If $T = A \multimap B$ then $\mathcal{E}(t, A \multimap B) = \mathcal{E}(t\mathcal{M}(A), B)$ and by Lemma 9 $\mathcal{M}(A)$ is a closed System $\mathcal{L}$ term of type A, thus by induction $\mathcal{E}(t\mathcal{M}(A), B) \longrightarrow^* I$. □

Explicit Copying. We have shown how to duplicate numbers in Section 4, 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. We proceed with an example before giving the general principle. Suppose that we want to write $\lambda x.\langle x, x\rangle$. This term can be *linearised*: $\lambda xy.\langle x, y\rangle$. If we now apply this term to two copies of the argument, we are done. Although we don't have the argument yet, we can write a System $\mathcal{L}$ term which will create these copies: $\lambda z.\mathsf{iter}\ (\mathsf{S}^2\ 0)\ (\lambda xy.\langle x, y\rangle)\ (\lambda x.xz)$.

Lemma 12 (Duplication Lemma). *If t is a closed System $\mathcal{L}$ term, then there is a System $\mathcal{L}$ term D such that $Dt = \langle t, t\rangle$.*

Proof. Let $D = \lambda z.\mathsf{iter}\ \mathsf{S}(\mathsf{S}\ 0)\ (\lambda xy.\langle x, y\rangle)\ (\lambda x.xz)$ then

$$\begin{aligned} Dt &\longrightarrow \mathsf{iter}\ \mathsf{S}(\mathsf{S}\ 0)\ (\lambda xy.\langle x, y\rangle)\ (\lambda x.xt) \\ &\longrightarrow^* (\lambda x.xt)((\lambda x.xt)(\lambda xy.\langle x, y\rangle)) \longrightarrow^* \langle t, t\rangle \end{aligned}$$

□

This result also applies to numbers, so we have two different ways of copying numbers in System $\mathcal{L}$.

5.1 Compilation

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

Definition 5. *System $\mathcal{T}$ types are translated into System $\mathcal{L}$ types using $\langle\cdot\rangle$ defined by:*

$$\begin{array}{llll} \langle\mathsf{Nat}\rangle & = \mathsf{Nat} & \langle\mathsf{Bool}\rangle & = \mathsf{Bool} \\ \langle A \to B\rangle & = \langle A\rangle \multimap \langle B\rangle & \langle A \times B\rangle & = \langle A\rangle \otimes \langle B\rangle \end{array}$$

If $\Gamma = x_1 \colon T_1, \ldots, x_n \colon T_n$ then $\langle\Gamma\rangle = x_1 \colon \langle T_1\rangle, \ldots, x_n \colon \langle T_n\rangle$.

Definition 6 (Compilation). *Let t be a System $\mathcal{T}$ term such that $\Gamma \vdash_{\mathcal{T}} t : T$. Its compilation into System $\mathcal{L}$ is defined as: $[x_1]\ldots[x_n]\langle t\rangle$ where $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, $n \geq 0$, we assume without loss of generality that the variables are processed in lexicographic order, and $\langle\cdot\rangle$, $[\cdot]\cdot$ are defined below. We abbreviate*

$\mathsf{iter}\ (\mathsf{S}^n\ 0)\ (\lambda x_1 \cdots x_n.t)\ (\lambda z.zx)$ *as* $C_x^{x_1,\ldots,x_n}\ t$, *and* $([x]t)[y/x]$ *as* $A_y^x t$.

$$
\begin{array}{ll}
\langle x \rangle & = x \\
\langle tu \rangle & = \langle t \rangle \langle u \rangle \\
\langle \lambda x.t \rangle & = \lambda x.[x]\langle t \rangle, \textit{if } x \in \mathsf{fv}(t) \\
& = \lambda x.\mathcal{E}(x, \langle A \rangle)\langle t \rangle, \textit{otherwise}, \qquad \textit{where } \Gamma \vdash_{\mathcal{T}} t : A \to B = T \textit{ (Lemma 1)} \\
\langle 0 \rangle & = 0 \\
\langle \mathsf{S}\ t \rangle & = \mathsf{S}\langle t \rangle \\
\langle \mathsf{iter}\ n\ u\ v \rangle & = \mathsf{iter}\ \langle n \rangle\ \langle u \rangle\ \langle v \rangle \\
\langle \mathsf{true} \rangle & = \mathsf{true} \\
\langle \mathsf{false} \rangle & = \mathsf{false} \\
\langle \mathtt{cond}\ n\ u\ v \rangle & = \mathtt{cond}\ \langle n \rangle\ \langle u \rangle\ \langle v \rangle \\
\langle \langle t, u \rangle \rangle & = \langle \langle t \rangle, \langle u \rangle \rangle \\
\langle \pi_1 t \rangle & = \mathtt{let}\ \langle x, y \rangle = \langle t \rangle\ \mathtt{in}\ \mathcal{E}(y, \langle B \rangle)x, \textit{ where } \Gamma \vdash_{\mathcal{T}} t : A \times B = T \textit{ (Lemma 1)} \\
\langle \pi_2 t \rangle & = \mathtt{let}\ \langle x, y \rangle = \langle t \rangle\ \mathtt{in}\ \mathcal{E}(x, \langle A \rangle)y, \textit{ where } \Gamma \vdash_{\mathcal{T}} t : A \times B = T \textit{ (Lemma 1)}
\end{array}
$$

and $[\cdot]\cdot$ *is defined as:*

$$
\begin{array}{ll}
[x](\mathsf{S}\ t) & = \mathsf{S}([x]t) \\
[x]x & = x \\
[x](\lambda y.t) & = \lambda y.[x]t \\
[x](tu) & = \begin{cases} C_x^{x_1,x_2}(A_{x_1}^x t)(A_{x_2}^x u) & x \in \mathsf{fv}(t), x \in \mathsf{fv}(u) \\ ([x]t)u & x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\ t([x]u) & x \in \mathsf{fv}(u), x \notin \mathsf{fv}(t) \end{cases} \\
[x](\mathsf{iter}\ n\ u\ v) & = \begin{cases} \mathsf{iter}\ [x]n\ u\ v & x \in \mathsf{fv}(n), x \notin \mathsf{fv}(uv) \\ \mathsf{iter}\ n\ [x]u\ v & x \notin \mathsf{fv}(nv), x \in \mathsf{fv}(u) \\ \mathsf{iter}\ n\ u\ [x]v & x \notin \mathsf{fv}(nu), x \in \mathsf{fv}(v) \\ C_x^{x,x_2}\mathsf{iter}\ (A_{x_1}^x n)\ (A_{x_2}^x u)\ v & x \in \mathsf{fv}(n) \cap \mathsf{fv}(u), x \notin \mathsf{fv}(v) \\ C_x^{x_1,x_3}\mathsf{iter}\ (A_{x_1}^x n)\ u\ (A_{x_3}^x v) & x \in \mathsf{fv}(n) \cap \mathsf{fv}(v), x \notin \mathsf{fv}(u) \\ C_x^{x_2,x_3}\mathsf{iter}\ n\ (A_{x_2}^x u)\ (A_{x_3}^x v) & x \notin \mathsf{fv}(n), x \in \mathsf{fv}(u) \cap \mathsf{fv}(v) \\ C_x^{x_1,x_2,x_3}\mathsf{iter}\ (A_{x_1}^x n)\ (A_{x_2}^x u)\ (A_{x_3}^x v) & x \in \mathsf{fv}(n) \cap \mathsf{fv}(u) \cap \mathsf{fv}(v) \end{cases}
\end{array}
$$

$[x](\mathtt{cond}\ n\ u\ v)$ *follows the same structure as* iter *above, replacing* iter *by* cond.

$$
[x]\langle t, u \rangle = \begin{cases} C_x^{x_1,x_2}\langle A_{x_1}^x t, A_{x_2}^x u \rangle, x \in \mathsf{fv}(t), x \in \mathsf{fv}(u) \\ \langle [x]t, u \rangle, x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\ \langle t, [x]u \rangle, x \in \mathsf{fv}(u), x \notin \mathsf{fv}(t) \end{cases}
$$

$$
[x](\mathtt{let}\ \langle y, z \rangle = t\ \mathtt{in}\ u) = \begin{cases} \mathtt{let}\ \langle y, z \rangle = [x]t\ \mathtt{in}\ u & x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\ \mathtt{let}\ \langle y, z \rangle = t\ \mathtt{in}\ [x]u & x \notin \mathsf{fv}(t), x \in \mathsf{fv}(u) \\ C_x^{x_1,x_2}(\mathtt{let}\ \langle y, z \rangle = A_{x_1}^x t\ \mathtt{in}\ A_{x_2}^x u) & x \in \mathsf{fv}(t), x \in \mathsf{fv}(u) \end{cases}
$$

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

As an example, we show the compilation of the combinators:
- $\langle \lambda x.x \rangle = \lambda x.x$
- $\langle \lambda xyz.xz(yz) \rangle = \lambda xyz.\mathsf{iter}\ \overline{2}\ (\lambda z_1 z_2.xz_1(yz_2))\ \lambda a.az$
- $\langle \lambda xy.x \rangle = \lambda xy.\mathcal{E}(y, B)x$

Lemma 13. *If t is a System $\mathcal{T}$ term, then:*

1. $\mathsf{fv}([x_1]\cdots[x_n]\langle t\rangle) = \mathsf{fv}(t)$.
2. $[x_1]\cdots[x_n]\langle t\rangle$ *is a valid System $\mathcal{L}$ term (satisfying the constraints in Table 1), if* $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$.

Proof. The first part is by induction on t using Lemma 9, and the second part by induction on t using the first part. □

We will now prove that the compilation produces a typable term in System $\mathcal{L}$. For this we will need a lemma in which we will use the type system for System $\mathcal{L}$ augmented with weakening and contraction rules for variables in a certain set X. Typing judgements in this system will be denoted $\Gamma \vdash_{L^{+X}} t : T$. We will denote $\Gamma_{|X}$ the restriction of Γ to the variables in X.

Lemma 14. *If $\Gamma \vdash_{\mathcal{T}} t : T$ and $\{x_1, \ldots, x_n\} \subseteq \mathsf{fv}(t)$ then*

1. $\langle \Gamma_{|\mathsf{fv}(t)} \rangle \vdash_{L^{+\mathsf{fv}(t)}} \langle t \rangle : \langle T \rangle$
2. $\langle \Gamma_{|\mathsf{fv}(t)} \rangle \vdash_{L^{+X}} [x_1]\ldots[x_n]\langle t\rangle \colon \langle T\rangle$ *implies*
   $\langle \Gamma_{|\mathsf{fv}(t)} \rangle \vdash_{L^{+X'}} [x][x_1]\ldots[x_n]\langle t\rangle \colon \langle T\rangle$ *where* $X = \mathsf{fv}(t) - \{x_1, \ldots, x_n\}$, $x \in X$, $X' = X - \{x\}$.

Proof. By simultaneous induction on t. □

Corollary 1. *If $\Gamma \vdash_{\mathcal{T}} t \colon T$ and $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ then*
$\langle \Gamma_{|\mathsf{fv}(t)} \rangle \vdash_{\mathcal{L}} [x_1]\ldots[x_n]\langle t\rangle \colon \langle T\rangle$.

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

Lemma 15 (Substitution). *Let t and w be System $\mathcal{L}$ terms such that $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, $n \geq 1$, and $\mathsf{fv}(w) = \varnothing$, then*
$([x_1]\ldots[x_n]\langle t\rangle)[\langle w\rangle/x_1] \longrightarrow^* [x_2]\ldots[x_n]\langle t[w/x_1]\rangle$.

Proof. By Lemma 13 (Part 1), $\mathsf{fv}(\langle w\rangle) = \varnothing$, and $x_1 \in \mathsf{fv}([x_1]\ldots[x_n]\langle t\rangle)$. We proceed by induction on t. □

Theorem 5 (Simulation). *Let t be a System $\mathcal{T}$ program, then: $t \Downarrow u \Rightarrow \langle t\rangle \longrightarrow^* \langle u\rangle$.*

Proof. By induction on $t \Downarrow u$. 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 $x \in \mathsf{fv}(t')$ then using Lemma 15:

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

Otherwise, using Lemmas 10 and 11:

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

Projection. By induction and Lemmas 10 and 11:

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

□

6 Conclusions

We have shown how to build a powerful calculus starting from the (very weak in terms of computational power) linear λ-calculus, by adding Booleans, numbers and linear iterators. We have seen that linear iterators can express much more than primitive recursive functions: the system has the computational power of System $\mathcal{T}$.

We have focused on the computational power of the linear calculus in this paper; there are other interesting aspects that 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: It is well-known that a Cartesian closed category (CCC) models the structure of the simply typed λ-calculus (i.e., a CCC is the internal language for the λ-calculus [17, 18]). The internal language of a symmetric monoidal closed category (SMCC) is the linear λ-calculus [20]. 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 natural question arises : what is the correspondence between CCC+NNO and SMCC+NNO?
- Does the technique extend to other typed λ-calculi, for instance the Calculus of Inductive Constructions [22]?

References

1. S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. A. Asperti. Light affine logic. In *Proc. Logic in Computer Science (LICS'98)*. IEEE Computer Society, 1998.
3. A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM Transactions on Computational Logic*, 2002.
4. 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.
5. 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.

6. 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.
7. J. Girard. Light linear logic. *Information and Computation*, 1998.
8. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
9. 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.
10. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
11. 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.
12. J. Hindley. BCK-combinators and linear lambda-terms have types. *Theoretical Computer Science (TCS)*, 64(1):97–105, April 1989.
13. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proc. Logic in Computer Science (LICS'99)*. IEEE Computer Society, 1999.
14. 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.
15. Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 2004.
16. U. D. 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 2005*, pages 366–375. IEEE Computer Society Press, June 2005.
17. 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.
18. J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics Vol. 7. Cambridge University Press, 1986.
19. I. Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.
20. I. Mackie, L. Román, and S. Abramsky. An internal language for autonomous categories. *Journal of Applied Categorical Structures*, 1(3):311–343, 1993.
21. M. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.
22. 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.
23. I. Phillips. Recursion theory. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 79–187. Oxford University Press, 1992.
24. K. Terui. Affine lambda-calculus and polytime strong normalization. In *Proc. Logic in Computer Science (LICS'01)*. IEEE Computer Society, 2001.