Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 57–74
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

The Power of Closed Reduction Strategies*

Sandra Alves[a,2,3] Maribel Fernández[b] Mario Florido[a,2]
Ian Mackie[b,c,1]

[a] *University of Porto, Department of Computer Science & LIACC, R. do Campo Alegre 823, 4150-180, Porto, Portugal*

[b] *King's College London, Department of Computer Science, Strand, London WC2R 2LS, U.K.*

[c] *LIX, École Polytechnique, 91128 Palaiseau Cedex, France*

Abstract

Closed reduction strategies in the λ-calculus restrict the reduction rules: the idea is that reductions can only take place when certain terms are closed (i.e. do not contain free variables). This has lead to various applications, such as an α-conversion free calculus of explicit substitutions, and an efficient abstract machine. The main contribution of this paper is a new application of this strategy to a linear version of Gödel's System T. We show that a linear System T with closed reduction offers a huge increase in expressive power over the usual linear systems, which are 'closed by construction' rather than 'closed at reduction'.

Keywords: linear calculi, iteration, System T, strategies.

1 Introduction

To prove the correctness of the Geometry of Interaction [7], Girard used a strategy for cut-elimination in linear logic [6] which restricts cut-elimination steps so that they 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 efficient in terms of the number of cut-elimination steps.

There are several translations of the λ-calculus into linear logic, which inspired the work on closed reduction in the λ-calculus [4,5]. Closed reductions avoid α-conversion by restricting β-reduction, so that only closed substitutions are gener-

* Research partially supported by the Treaty of Windsor Grant: "Linearity: Programming Languages and Implementations".

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

[2] Partially supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *FEDER/POSI*.

[3] Programa Gulbenkian de Estímulo à Investigação.

doi:10.1016/j.entcs.2007.02.047

ated. In contrast with standard weak strategies, which also avoid α-conversion, closed reduction strategies allow β-reductions to take place under lambdas, which means that more sharing of computation can be achieved. Note that we use the word *strategy* to refer to a restriction in the application of reduction rules, for instance, the so-called weak strategies in the λ-calculus do not allow β-reductions inside lambda-abstractions, and the closed reduction systems defined in this paper will not allow the β-rule to apply when the argument is open.

Closed reductions have interesting properties: in addition to offering efficient reduction strategies, applications such as new calculi of explicit substitution were obtained; see [5] for more details. The purpose of this paper is to examine this family of strategies further. More precisely, in this paper we will analyse the computational power of *linear λ-calculi with iterators*, by using closed reductions (whereas the closed reduction systems studied in [4,5] are not linear and do not contain iterators in the syntax).

In [2] we defined a linear version of Gödel's System T with closed reductions, which we called System $\mathcal{L}$. We showed that this linear system has all the computational power of System T. This result is surprising because usual definitions of linear systems with iterators are strictly less powerful than System T [11,9].

In this paper, we claim that the use of closed reduction is a key to the power of System $\mathcal{L}$. To support this claim, we analyse the interplay between linearity and closed reduction, and compare the computational power of linear systems with and without closed reduction.

We will define two linear systems: $\mathcal{L}^{\mathbb{N}}$ and $\mathcal{L}_0^{\mathbb{N}}$. Both systems are extensions of the linear λ-calculus [1] with numbers, booleans, pairs of natural numbers, and an iterator. System $\mathcal{L}^{\mathbb{N}}$ has the same syntax as System $\mathcal{L}$ [2], and as System $\mathcal{L}$ it uses the closed reduction strategy. However, its type system is more restrictive than System $\mathcal{L}$'s. We will show that System $\mathcal{L}^{\mathbb{N}}$ can encode all the primitive recursive functions and more general functions such as Ackermann's. On the other hand, System $\mathcal{L}_0^{\mathbb{N}}$ does not restrict to closed reduction strategies, but, to be linear, it has to restrict the set of terms. Actually, System $\mathcal{L}_0^{\mathbb{N}}$ can be seen as a subsystem of Dal Lago's linear language $H(\varnothing)$ [11], albeit with a different syntax. System $\mathcal{L}_0^{\mathbb{N}}$ can encode only primitive recursive functions (Ackermann's function is not definable), therefore this system is strictly weaker than System $\mathcal{L}^{\mathbb{N}}$, and hence also weaker than System $\mathcal{L}$.

In the next section we recall some background material. In Section 3 we define the linear systems $\mathcal{L}^{\mathbb{N}}$ and $\mathcal{L}_0^{\mathbb{N}}$, and in Section 4 we demonstrate that we can encode all the primitive recursive functions in these calculi. In Section 5 we show that System $\mathcal{L}^{\mathbb{N}}$ goes considerably beyond this class of functions. Finally we conclude the paper in Section 6.

2 Background: Closed Reduction, Linear Systems

Çağman and Hindley [3] observed that α-conversion can be avoided if β-redexes are closed (i.e. $(\lambda x.t)u$ does not contain free variables). However, this is a strong

restriction on the application of the β-reduction rule, and the resulting calculus is very weak. In [4,5] closed reduction strategies were investigated which are less restrictive than this. The motivation for this study was to understand efficiency issues in addition to finding calculi that were free from α-conversion.

Two different versions of closed reduction were studied, based around the following two options:

$$(\lambda x.t)u \to t[u/x] \quad \text{if } \mathsf{fv}(\lambda x.t) = \varnothing$$

$$(\lambda x.t)u \to t[u/x] \quad \text{if } \mathsf{fv}(u) = \varnothing$$

which correspond to closed function (**cf**) and closed argument (**ca**) respectively. In both cases, there are a number of variants that lead to calculi with different properties. Substitution is taken to be explicit in these systems (see [5] for more details).

The closed argument strategy was used in [2] to define an extension of the linear λ-calculus [1] with natural numbers, booleans, linear pairs, linear conditionals and a linear iterator (and with implicit rather than explicit substitution). This linear version of Gödel's System T was called System $\mathcal{L}$.

In [12] it was shown that the linear λ-calculus is the internal language for symmetric monoidal closed categories (the analogous result to the λ-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 type-environment, that is, iterators are "closed by *construction*":

$$\frac{\Gamma \vdash n : \mathbb{N} \quad \Delta \vdash b : A \quad \vdash f : A \multimap A}{\Gamma, \Delta \vdash \mathsf{iter}\ n\ b\ f}$$

In the same line, the linear System T of [11], called $H(\varnothing)$, only allows the construction of iterators on closed functions, and can only encode primitive recursive functions. On the other hand, System $\mathcal{L}$ has all the power of the full System T. To understand what gives these two linear versions of System T so different properties, in the following sections we will define two linear systems, one which allows us to build iterators on functions with free variables, but requires that reduction takes place only after the functions become closed, and another that does not use closed reduction, but requires iterators to be closed by construction.

3 Linear Systems with and without Closed Reduction

In this section we will define two linear systems: System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}^{\mathbb{N}}_0$. Both systems extend the linear λ-calculus with booleans, numbers, pairs of natural numbers, and an iterator. While System $\mathcal{L}^{\mathbb{N}}_0$ has the usual open β-reduction rule but, when building an iterator, requires the iterated function to be closed (therefore avoiding copying of free variables), System $\mathcal{L}^{\mathbb{N}}$ uses a closed reduction strategy [5,8] and allows the use of open functions in iterators.

We start by defining the syntax and the set of types, which will be common to the two systems (see also [2]).

3.1 Linear Terms and Types

The linear λ-terms are terms from the λ-calculus restricted in the following way ($\mathsf{fv}(t)$ denotes the set of free variables of t):

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

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: numbers, booleans and pairs, preserving syntactic linearity. We use the following syntax:

- *Pairs*:

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

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

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

- *Booleans*: true and false, and a conditional:

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

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

- *Numbers*: 0 and S, and an iterator:

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

We follow standard notational conventions in the sequel: for instance, we write $\lambda xy.t$ instead of $\lambda x.\lambda y.t$, application is left-associative and we use brackets only when there is ambiguity. As an abbreviation, we write $\mathsf{S}^n 0$ to denote n applications of S to 0.

Table 1 summarises the syntax of System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}^{\mathbb{N}}_0$, showing the term construction, variable constraints and free variables of terms.

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

Table 1
Syntax of Terms

3.1.1 Types

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

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

That is, we consider two base types (natural numbers and booleans), linear arrows, and linear products on natural numbers.

3.2 System $\mathcal{L}^{\mathbb{N}}$

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

The dynamics of the system is given by a set of conditional reduction rules (which can be seen as a higher-order membership conditional rewrite system, see [14,15]). The conditions on the rewrite rules restrict the rewrite relation, ensuring that *Beta* only applies to redexes where the argument is a closed term (which implies that α-conversion is not needed to implement substitution), and only closed functions are iterated.

Definition 3.1 (Closed Reduction) Table 2 gives the reduction rules for System $\mathcal{L}^{\mathbb{N}}$, substitution is a meta-operation defined as usual. Reductions can take place in any context. We use $\longrightarrow$ to denote the one-step reduction relation, and $\longrightarrow^*$ for its reflexive and transitive closure.

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

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

to a closed argument reduction strategy (ca, see [5]). Also note that *Iter* rules cannot be applied if the function v is open.

System $\mathcal{L}^{\mathbb{N}}$'s syntax and reduction rules are the same as System $\mathcal{L}$'s [2], as a consequence we inherit from System $\mathcal{L}$ the following properties, for the untyped calculus:

Lemma 3.2 (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 3.3 (Correctness of $\longrightarrow$) *Let t be a valid term, and $t \longrightarrow u$, then:*

(i) $\mathsf{fv}(t) = \mathsf{fv}(u)$;

(ii) *u is a valid term.*

Lemma 3.4 (Confluence) *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 $t_2 \longrightarrow^* t_3$.*

We associate types to terms in System $\mathcal{L}^{\mathbb{N}}$ using the typing rules given in Figure 1.

Since we are in a linear system, we do not have Weakening and Contraction rules. The only structural rule in Figure 1 is Exchange. For the same reason, the logical rules split the typing 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 $\mathbb{N}$ and that v and u are compatible.

Note that we allow the typing of $\mathsf{iter}\ t\ u\ v$ even if v is open (in contrast with [11,9]), but we do not allow reduction until v is closed. We will show later that this feature gives System $\mathcal{L}^{\mathbb{N}}$ more power (whereas systems that do not allow building an iterator with v open are strictly weaker).

Subject Reduction for System $\mathcal{L}^{\mathbb{N}}$ can be proved as for System $\mathcal{L}$ [2].

Theorem 3.5 (Subject Reduction) *If $\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : A$ and $t \longrightarrow u$, then*

$$\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} u : A.$$

Axiom and **Structural Rule**:

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

Logical Rules:

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

$$\frac{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : \mathbb{N} \qquad \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} u : \mathbb{N}}{\Gamma, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} \langle t, u \rangle : \mathbb{N} \otimes \mathbb{N}}\,(\otimes\mathsf{Intro})$$

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

Numbers:

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

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

Booleans:

$$\frac{}{\vdash_{\mathcal{L}^{\mathbb{N}}} \mathsf{true} : \mathbb{B}}\,(\mathsf{True}) \qquad \frac{}{\vdash_{\mathcal{L}^{\mathbb{N}}} \mathsf{false} : \mathbb{B}}\,(\mathsf{False})$$

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

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

Proof. (Sketch) By induction on the type derivation $\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : A$, using a Substitution Lemma: If $\Gamma, x : A \vdash_{\mathcal{L}^{\mathbb{N}}} u : B$ and $\Delta \vdash_{\mathcal{L}^{\mathbb{N}}} v : A$ (where $\mathsf{fv}(u) \cap \mathsf{fv}(v) = \varnothing$) then $\Gamma, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} u[v/x] : B$. □

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

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

Theorem 3.6 (Strong Normalisation) *If $\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : T$, then t is strongly normalisable.*

3.3 System $\mathcal{L}_0^{\mathbb{N}}$

The set of terms for System $\mathcal{L}_0^{\mathbb{N}}$ is built in the same way as for System $\mathcal{L}^{\mathbb{N}}$, except that when building an iterator, we don't allow the iterated function to be an open

term. Thus iterators in this system have the following definition (note the additional constraint $\mathsf{fv}(v) = \varnothing$):

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

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

Definition 3.7 (Reduction) Table 3 gives the reduction rules for System $\mathcal{L}_0^{\mathbb{N}}$; substitution is a meta-operation defined as usual. Reductions can take place in any context.

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

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

Correctness of Substitution is proved as for System $\mathcal{L}$ [2], but α-conversion must be used in substitution whenever necessary. Note that α-conversion was not needed in System $\mathcal{L}$ and therefore in System $\mathcal{L}^{\mathbb{N}}$, because all the substitutions take a closed term.

Lemma 3.8 (Correctness of Substitution) *Let t and u be valid terms, such that $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$ and $x \in \mathsf{fv}(t)$, then $t[u/x]$ is valid.*

Proof. (Sketch) Straightforward induction on the structure of t, showing that substitution preserves the variable constraints on terms. □

Lemma 3.9 (Correctness of $\longrightarrow$) *Let t be a valid term, and $t \longrightarrow u$, then:*

(i) $\mathsf{fv}(t) = \mathsf{fv}(u)$;

(ii) *u is a valid term.*

Proof. (Sketch) By structural induction on t, showing that reduction preserves the variable constraints on terms. Note that 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 $\mathsf{iter}\ t\ u\ v$, 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. □

Note that in System $\mathcal{L}^{\mathbb{N}}$ we do not have the constraint on the iterator term, but we have a condition in the reduction rules for iterators, which also guarantees that reducing an iterator will either copy or erase a closed term.

In Figure 2 we show how types are assigned to terms in System $\mathcal{L}_0^{\mathbb{N}}$. The only difference between the rules for this system and for System $\mathcal{L}^{\mathbb{N}}$ is in the (Iter) rule, where the typing context for the iterated function is always empty, which forces the iterated function to be closed.

Axiom and **Structural Rule**:

$$\frac{}{x : A \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : A}\ (\mathsf{Axiom}) \qquad \frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : C}\ (\mathsf{Exchange})$$

Logical Rules:

$$\frac{\Gamma, x : A \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : B}{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda x.t : A \multimap B}\ (\multimap\mathsf{Intro}) \qquad \frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : A \multimap B \qquad \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : A}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} tu : B}\ (\multimap\mathsf{Elim})$$

$$\frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{N} \qquad \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : \mathbb{N}}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle t, u \rangle : \mathbb{N} \otimes \mathbb{N}}\ (\otimes\mathsf{Intro})$$

$$\frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{N} \otimes \mathbb{N} \qquad x : \mathbb{N}, y : \mathbb{N}, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : C}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle x, y \rangle = t\ \mathtt{in}\ u : C}\ (\otimes\mathsf{Elim})$$

Numbers:

$$\frac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} 0 : \mathbb{N}}\ (\mathsf{Zero}) \qquad \frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} n : \mathbb{N}}{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ n : \mathbb{N}}\ (\mathsf{Succ})$$

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

Booleans:

$$\frac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{true} : \mathbb{B}}\ (\mathsf{True}) \qquad \frac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{false} : \mathbb{B}}\ (\mathsf{False})$$

$$\frac{\Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{B} \quad \Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : A \quad \Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} v : A}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{cond}\ t\ u\ v : A}\ (\mathsf{Cond})$$

Fig. 2. Type System for System $\mathcal{L}_0^{\mathbb{N}}$

As before, Subject Reduction for System $\mathcal{L}_0^{\mathbb{N}}$ can be proved as for System $\mathcal{L}$ [2].

Theorem 3.10 (Subject Reduction) *If* $\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : A$ *and* $t \longrightarrow u$, *then*

$$\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : A.$$

Theorem 3.11 (Strong Normalisation) *If* $\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : T$, *then* t *is strongly normalisable.*

Proof. 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 normalising. □

Lemma 3.12 (Confluence) *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* $t_2 \longrightarrow^* t_3$.

Proof. 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 Newmann's Lemma [13]). Moreover, we can apply directly Klop's result [10] to the untyped calculus because the system is orthogonal (that is, left-linear and non-overlapping).□

4 Primitive Recursive Functions

Based on the results obtained for System $\mathcal{L}$ [2], in this section we show how we can define the primitive recursive functions in both System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}_0^{\mathbb{N}}$. We choose to present an encoding that satisfies the term conditions of System $\mathcal{L}_0^{\mathbb{N}}$, since these are more restrictive than those of System $\mathcal{L}^{\mathbb{N}}$. In the next section we show that in System $\mathcal{L}^{\mathbb{N}}$ we can encode substantially more than primitive recursive functions.

We start by recalling the definition of primitive recursive function.

Definition 4.1 A function $f : \mathbb{N}^n \to \mathbb{N}$ is primitive recursive if it can be defined using: the natural numbers; the projections: $\pi_i^n(x_1, \ldots, x_n) = x_i$ $(1 \leq i \leq n)$; composition; and the primitive recursive scheme, which allows us to define a recursive function h using two auxiliary (primitive recursive) functions f, g:

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

4.1 Erasing linearly

Although System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}_0^{\mathbb{N}}$ are linear calculi, we can erase numbers. In particular, we can define the projection functions on $\mathbb{N}^2$ $\mathsf{fst}, \mathsf{snd} : \mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N}$ as follows:

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

Lemma 4.2 *For any numbers* a *and* b, $\mathsf{fst}\langle a, b\rangle \longrightarrow^* a$ *and* $\mathsf{snd}\langle a, b\rangle \longrightarrow^* b$.

$$\dfrac{\dfrac{}{x : \mathbb{N} \otimes \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N} \otimes \mathbb{N}} \quad \dfrac{\dfrac{}{v : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} v : \mathbb{N}} \quad \dfrac{}{u : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : \mathbb{N}} \quad \dfrac{\dfrac{}{z : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} z : \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda z.z : \mathbb{N} \multimap \mathbb{N}}}{u : \mathbb{N}, v : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ v\ u\ \lambda z.z : \mathbb{N}}}{x : \mathbb{N} \otimes \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle u, v\rangle = x\ \mathtt{in}\ \mathsf{iter}\ v\ u\ \lambda z.z : \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda x.\mathtt{let}\ \langle u, v\rangle = x\ \mathtt{in}\ \mathsf{iter}\ v\ u\ \lambda z.z : (\mathbb{N} \otimes \mathbb{N}) \multimap \mathbb{N}}$$

Fig. 3. Typing of fst

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

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

□

Note that fst and snd are valid typable terms in System $\mathcal{L}_0^{\mathbb{N}}$ (see Figure 3), and they are closed terms.

4.2 Copying linearly

We can also copy natural numbers in these linear calculi. For this, we define a function $C : \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$ that given a number n returns a pair $\langle n, n\rangle$:

$$C = \lambda x.\mathsf{iter}\ x\ \langle 0, 0\rangle\ (\lambda x.\mathtt{let}\ \langle a, b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle)$$

Lemma 4.3 *For any number n, $C\ n \longrightarrow^* \langle n, n\rangle$.*

In the derivation below, s is the term $(\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle)$

$$\dfrac{\dfrac{}{x : \mathbb{N}\otimes\mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N}\otimes\mathbb{N}} \qquad \dfrac{\dfrac{\dfrac{}{a : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} a : \mathbb{N}}}{a : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ a : \mathbb{N}} \qquad \dfrac{\dfrac{}{b : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} b : \mathbb{N}}}{b : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ b : \mathbb{N}}}{a : \mathbb{N}, b : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle : \mathbb{N}\otimes\mathbb{N}}}{x : \mathbb{N}\otimes\mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle : \mathbb{N}\otimes\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle) : (\mathbb{N}\otimes\mathbb{N}) \multimap (\mathbb{N}\otimes\mathbb{N})}$$

$$\dfrac{\dfrac{}{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N}} \qquad \dfrac{\dfrac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} 0 : \mathbb{N}} \qquad \dfrac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} 0 : \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle 0,0\rangle : \mathbb{N}\otimes\mathbb{N}} \qquad \vdots \quad \vdash_{\mathcal{L}_0^{\mathbb{N}}} s : (\mathbb{N}\otimes\mathbb{N}) \multimap (\mathbb{N}\otimes\mathbb{N})}{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ x\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle) : \mathbb{N}\otimes\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda x.\mathsf{iter}\ x\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle) : \mathbb{N} \multimap (\mathbb{N}\otimes\mathbb{N})}$$

Fig. 4. Typing of C

Proof. By induction on n.

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

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

□

Again, C is valid in System $\mathcal{L}_0^{\mathbb{N}}$ (see Figure 4), and a closed term.

4.3 Primitive Recursive Scheme

We have already shown we can project, and of course we have composition. We now show how to encode, using iterators, a function h defined by primitive recursion from f and g (see Definition 4.1).

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 the closed functions $G : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}$ and $F : \mathbb{N} \multimap \mathbb{N}$, representing the primitive recursive functions g and f, let g' be the term:

$$\lambda y.\lambda z.\mathtt{let}\ \langle z_1, z_2\rangle = C\ z\ \mathtt{in}\ Gz_1(yz_2) : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})$$

then $h(x,n)$ is defined by the term $(\mathsf{iter}\ n\ F\ g')\ x : \mathbb{N}$, (see Figure 5). Note that the encoding of h is a closed term. This term is valid because F is closed (by assumption), and g' is closed since G is a closed term (by assumption).

Lemma 4.4 *For any numbers x and n, $(\mathsf{iter}\ n\ F\ g')\ x$ reduces to the number $h(x,n)$.*

Proof. By induction, using Lemma 4.3 to copy numbers:

$$(\mathsf{iter}\ 0\ F\ g')\ x \longrightarrow (F\ x) = h(x,0)$$

$$\begin{aligned} (\mathsf{iter}\ (\mathsf{S}^{n+1}\ 0)\ F\ g')\ x &\longrightarrow g'\ (\mathsf{iter}\ (\mathsf{S}^n\ 0)\ F\ g')\ x \\ &\longrightarrow^* \mathtt{let}\ \langle z_1, z_2\rangle = \langle x, x\rangle\ \mathtt{in}\ G\ z_1\ ((\mathsf{iter}\ (\mathsf{S}^n\ 0)\ F\ g')\ z_2) \\ &\longrightarrow G\ x\ ((\mathsf{iter}\ (\mathsf{S}^n\ 0)\ F\ g')\ x) \\ &\longrightarrow^* G\ x\ h(x,n) \text{ by induction hypothesis} \\ &= h(x,n+1). \end{aligned}$$

□

Now to encode the standard primitive recursive scheme, which has an extra n in the last equation (see Definition 4.1), 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 $\mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}$):

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

where

$$\begin{aligned} s &= \mathsf{iter}\ n_2\ (\lambda x_1.\mathsf{fst}\langle F, x_1\rangle)\ s' \\ s' &= \lambda y x_2 z. \\ &\quad \mathtt{let}\ \langle z_1, z_2\rangle = C\ z\ \mathtt{in}\ (\mathtt{let}\ \langle w_1, w_2\rangle = C\ x_2\ \mathtt{in}\ G\ z_1\ (y\ (\mathsf{pred}\ w_1)\ z_2)\ w_2) \end{aligned}$$

$$\dfrac{\dfrac{\vdots}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} G : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}} \quad \dfrac{}{z_1 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} z_1 : \mathbb{N}}}{z_1 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} Gz_1 : \mathbb{N} \multimap \mathbb{N}} \qquad \dfrac{\dfrac{}{y : \mathbb{N} \multimap \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} y : \mathbb{N} \multimap \mathbb{N}} \quad \dfrac{}{z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} z_2 : \mathbb{N}}}{y : \mathbb{N} \multimap \mathbb{N}, z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} yz_2 : \mathbb{N}}$$

$$\dfrac{}{y : \mathbb{N} \multimap \mathbb{N}, z_1 : \mathbb{N}, z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} Gz_1(yz_2) : \mathbb{N}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{z : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} C\ z : \mathbb{N} \otimes \mathbb{N}} \quad \dfrac{\vdots}{y : \mathbb{N} \multimap \mathbb{N}, z_1 : \mathbb{N}, z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} Gz_1(yz_2) : \mathbb{N}}}{y : \mathbb{N} \multimap \mathbb{N}, z : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle z_1, z_2 \rangle = C\ z\ \mathtt{in}\ Gz_1(yz_2) : \mathbb{N}}}{y : \mathbb{N} \multimap \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda z.\mathtt{let}\ \langle z_1, z_2 \rangle = C\ z\ \mathtt{in}\ Gz_1(yz_2) : \mathbb{N} \multimap \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda yz.\mathtt{let}\ \langle z_1, z_2 \rangle = C\ z\ \mathtt{in}\ Gz_1(yz_2) : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N}} \quad \dfrac{\vdots}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} F : \mathbb{N} \multimap \mathbb{N}} \quad \dfrac{\vdots}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} g' : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})}}{n : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ n\ F\ g' : \mathbb{N} \multimap \mathbb{N} \quad x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N}}}{n : \mathbb{N}, x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} (\mathsf{iter}\ n\ F\ g')x : \mathbb{N}}$$

Fig. 5. Typing of functions defined by primitive recursion

Let pred be the encoding of the predecessor function defined as:

$$\mathsf{pred} = \lambda n.\mathsf{fst}(\mathsf{iter}\ n\ \langle 0, 0\rangle\ (\lambda x.\mathtt{let}\ \langle t, u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t, \mathsf{S}\ u\rangle))$$

It can be easily verified by induction that the predecessor function is correct, that is:

$$\begin{aligned} \mathsf{pred}\ 0 &= 0 \\ \mathsf{pred}\ (\mathsf{S}\ n) &= n \end{aligned}$$

Also, the predecessor function is a closed typable function in System $\mathcal{L}_0^{\mathbb{N}}$ (see Figure 6).

5 Beyond Primitive Recursion

In this section we show that it is possible to encode more than primitive recursive functions in System $\mathcal{L}^{\mathbb{N}}$, by giving the encoding of a well-known non primitive recursive function: Ackermann's function.

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

In the derivation below, f is the term $\lambda x.\mathtt{let}\ \langle t,u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t,\mathsf{S}\ u\rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x:\mathbb{N}\otimes\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} C(\mathsf{snd}\ x):\mathbb{N}\otimes\mathbb{N}} \qquad \dfrac{\dfrac{}{t:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} t:\mathbb{N}} \qquad \dfrac{\dfrac{}{u:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} u:\mathbb{N}}}{u:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ u:\mathbb{N}}}{t:\mathbb{N},u:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle t,\mathsf{S}\ u\rangle:\mathbb{N}\otimes\mathbb{N}}}{x:\mathbb{N}\otimes\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle t,u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t,\mathsf{S}\ u\rangle:\mathbb{N}\otimes\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda x.\mathtt{let}\ \langle t,u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t,\mathsf{S}\ u\rangle:(\mathbb{N}\otimes\mathbb{N})\multimap(\mathbb{N}\otimes\mathbb{N})}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{n:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} n:\mathbb{N}} \qquad \dfrac{\vdots}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle 0,0\rangle:\mathbb{N}\otimes\mathbb{N}} \qquad \dfrac{\vdots}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} f:(\mathbb{N}\otimes\mathbb{N})\multimap(\mathbb{N}\otimes\mathbb{N})}}{n:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ n\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle t,u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t,\mathsf{S}\ u\rangle):\mathbb{N}\otimes\mathbb{N}}}{n:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{fst}(\mathsf{iter}\ n\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle t,u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t,\mathsf{S}\ u\rangle)):\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{fst}(\mathsf{iter}\ n\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle t,u\rangle = C(\mathsf{snd}\ x)\ \mathtt{in}\ \langle t,\mathsf{S}\ u\rangle)):\mathbb{N}\multimap\mathbb{N}}$$

Fig. 6. Typing of predecessor

In a higher-order functional language, it can be defined as follows:
Let $\mathsf{succ} = \lambda x.\mathsf{S}\ x : \mathbb{N}\multimap\mathbb{N}$, then $ack(m,n) = a\ m\ n$ where:

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

$$\begin{aligned} A\ g\ 0 &= g(\mathsf{S}\ 0) \\ A\ g\ (\mathsf{S}\ n) &= g(A\ g\ n) \end{aligned}$$

Lemma 5.1 *Both definitions are equivalent: $a\ x\ y = ack(x,y)$, for all numbers x,y.*

Proof. By induction on x, using Lemma 5.2 below:

- The case $x = 0$ is trivial: $a\ 0\ y = \mathsf{S}\ y = ack(0,y)$.
- By definition, $a\ (\mathsf{S}\ n) = A\ (a\ n)$, and by hypothesis this is $A(\lambda y.ack(n,y))$. Therefore by Lemma 5.2 below, $a\ (\mathsf{S}\ n)\ z = ack(\mathsf{S}\ n, z)$.

□

Lemma 5.2 *If $g = \lambda y.ack(x,y)$ then $A\ g\ n = ack(\mathsf{S}\ x, n)$.*

Proof. By induction on n.

- $A\ g\ 0 = g(\mathsf{S}\ 0) = ack(x,\mathsf{S}\ 0) = ack(\mathsf{S}\ x, 0)$.

- $A\ g\ (\mathsf{S}\ n) = g\ (A\ g\ n) = g\ (ack(\mathsf{S}\ x, n)) = ack(x, ack(\mathsf{S}\ x, n)) = ack(\mathsf{S}\ x, \mathsf{S}\ n)$.

□

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

$$a\ = \lambda n.\mathsf{iter}\ n\ \mathsf{succ}\ A : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}$$
$$A = \lambda g\ n.\mathsf{iter}\ (\mathsf{S}\ n)\ (\mathsf{S}\ 0)\ g : (\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{N} \multimap \mathbb{N}$$

We show by induction that this encoding is correct (we assume n is a number and g a closed function):

$$\begin{array}{lll} a\ 0 & \longrightarrow & \mathsf{iter}\ 0\ \mathsf{succ}\ A \\ & \longrightarrow & \mathsf{succ} \\ a\ (\mathsf{S}\ n) & \longrightarrow & \mathsf{iter}\ (\mathsf{S}\ n)\ \mathsf{succ}\ A \\ & \longrightarrow & A(\mathsf{iter}\ n\ \mathsf{succ}\ A) = A(a\ n) \end{array}$$

$$\begin{array}{lll} A\ g\ 0 & \longrightarrow^* & \mathsf{iter}\ (\mathsf{S}\ 0)\ (\mathsf{S}\ 0)\ g \\ & \longrightarrow^* & g(\mathsf{S}\ 0) \\ A\ g\ (\mathsf{S}\ n) & \longrightarrow^* & \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\ n))\ (\mathsf{S}\ 0)\ g \\ & \longrightarrow & g\ (\mathsf{iter}\ (\mathsf{S}\ n)\ (\mathsf{S}\ 0)\ g) = g(A\ g\ n) \end{array}$$

Then Ackermann's function can be defined in System $\mathcal{L}^{\mathbb{N}}$ (see Figure 7) as:

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

The correctness of this encoding follows directly from the lemmas above.

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

Every function in System $\mathcal{L}_0^{\mathbb{N}}$ can be defined in the system $H(\varnothing)$ studied in [11]: the syntax is different but the typing rules are equivalent (it is possible to define an encoding from terms/typings in System $\mathcal{L}_0^{\mathbb{N}}$ to terms/typings in $H(\varnothing)$). It has been proved (see [11] for details) that Ackermann's function cannot be represented in $H(\varnothing)$, therefore it cannot be represented in System $\mathcal{L}_0^{\mathbb{N}}$ either. Thus, System $\mathcal{L}_0^{\mathbb{N}}$ is strictly less powerful than System $\mathcal{L}^{\mathbb{N}}$.

6 Conclusions and Future Work

Closed reduction strategies impose strong constraints on the application of reduction rules, but despite this fact, they can simulate both call-by-name and call-by-value

$$\dfrac{\dfrac{\dfrac{}{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N}}}{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ x : \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{succ} = \lambda x.\mathsf{S}\ x : \mathbb{N} \multimap \mathbb{N}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : \mathbb{N}}}{u : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ u : \mathbb{N}} \quad \dfrac{\dfrac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} 0 : \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ 0 : \mathbb{N}} \quad \dfrac{}{g : \mathbb{N} \multimap \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} g : \mathbb{N} \multimap \mathbb{N}}}{g : \mathbb{N} \multimap \mathbb{N}, u : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g : \mathbb{N}}}{g : \mathbb{N} \multimap \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda u.\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g : (\mathbb{N} \multimap \mathbb{N})}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} A = \lambda g u.\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{m : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} m : \mathbb{N}} \quad \begin{array}{c}\vdots\\ \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{succ} : \mathbb{N} \multimap \mathbb{N}\end{array} \quad \begin{array}{c}\vdots\\ \vdash_{\mathcal{L}_0^{\mathbb{N}}} A : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})\end{array}}{m : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ m\ \mathsf{succ}\ A : \mathbb{N} \multimap \mathbb{N}} \quad \dfrac{}{n : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} n : \mathbb{N}}}{m : \mathbb{N}, n : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} (\mathsf{iter}\ m\ \mathsf{succ}\ A)\ n : \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda m\ n.(\mathsf{iter}\ m\ \mathsf{succ}\ A)\ n : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}}$$

Fig. 7. Typing of Ackermann's function

evaluations in the λ-calculus (as shown in [4]), and also more efficient evaluations (since reductions can take place under abstractions and thus achieve more sharing of computations); similar results hold for PCF (see [5]).

In this paper we have shown that in the case of a linear λ-calculus with iterators, the use of closed reduction strategies has another benefit: not only we can gain in efficiency, but also we gain in computational power, thanks to the fact that we can relax the constraints on the construction of iterator terms.

Acknowledgement

We are grateful to the anonymous referees for their comments, and to Bernhard Gramlich for pointing out relevant references.

References

[1] S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of linear functions. In *Proceedings of*

Computer Science Logic (CSL'06), volume 4207 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, September 2006.

[3] N. Çağman and J. R. Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1–2):239–249, 1998.

[4] M. Fernández and I. Mackie. Closed reduction in the λ-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.

[5] 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.

[6] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[7] 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.

[8] 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.

[9] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proc. Logic in Computer Science (LICS'99)*. IEEE Computer Society, 1999.

[10] J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[11] 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.

[12] I. Mackie, L. Román, and S. Abramsky. An internal language for autonomous categories. *Journal of Applied Categorical Structures*, 1(3):311–343, 1993.

[13] M. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[14] Y. Toyama. Confluent term rewriting systems with membership conditions. In *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems, CTRS'87, Orsay, France*, volume 308 of *LNCS*, pages 228–241. Springer-Verlag, 1988.

[15] J. Yamada. Confluence of terminating membership conditional TRS. In *Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems, CTRS'92, Pont--Mousson, France*, volume 656 of *LNCS*, pages 378–392. Springer-Verlag, 1993.