Soft lambda-calculus: a language for polynomial time computation.

Patrick Baillot, Virgile Mogbil

Soft lambda-calculus: a language for polynomial time computation

Patrick Baillot and Virgile Mogbil

Laboratoire d'Informatique de Paris-Nord UMR 7030 CNRS
Université Paris XIII - Institut Galilée, 99 Avenue Jean-Baptiste Clément,
93430 Villetaneuse, France.
{patrick.baillot,virgile.mogbil}@lipn.univ-paris13.fr *

Abstract. Soft linear logic ([Lafont02]) is a subsystem of linear logic characterizing the class PTIME. We introduce *Soft lambda-calculus* as a calculus typable in the intuitionistic and affine variant of this logic. We prove that the (untyped) terms of this calculus are reducible in polynomial time. We then extend the type system of Soft logic with recursive types. This allows us to consider non-standard types for representing lists. Using these datatypes we examine the concrete expressiveness of Soft lambda-calculus with the example of the insertion sort algorithm.

1 Introduction

The advent of global computing has increased the need for formal bounds on the use of resources by programs. This issue arises is a variety of situations like when running code originating from untrusted source or in settings where memory or time is constrained, for instance in embedded or synchronous systems.

Some cornerstones have been laid by the work in *Implicit Computational Complexity* (ICC) carried out by several authors since the 1990s ([16, 17, 6] among others). This field aims at studying languages and calculi in which all programs fall into a given complexity class. The most studied case has been that of deterministic polynomial time complexity (PTIME class).

We can in particular distinguish two important lines of work. The first one deals with primitive recursion and proposes restrictions on primitive recursion such that the functions definable are those of PTIME: this is the approach of *safe* or *ramified recursion* ([6, 16]) and subsequent extensions ([13, 7]). Another line is that of Linear logic (LL)([9]). The Curry-Howard correspondence allows us to see proofs in this logic as programs. Linear logic provides a way of controlling duplication of arguments thanks to specific modalities (called *exponentials*). It is possible to consider variants of LL with alternative, stricter rules for modalities, for which all proofs-programs can be run in polynomial time.

Light linear logic, introduced by Girard ([10]) is one of these systems. It has been later simplified by Asperti into Light affine logic ([3]) which allows

* Work partially supported by Action Spécifique CNRS *Méthodes formelles pour la Mobilité* and ACI Sécurité Informatique CRISS.

for arbitrary erasing. However formulas in this system are quite complicated as there are two modalities, instead of just one in Intuitionistic linear logic, which makes programming delicate (see [2, 4]). More recently Lafont has introduced Soft linear logic (SLL) ([15]), a simpler system which uses the same language of formulas as Linear logic and is polytime. It can in fact be seen as a subsystem of Linear logic or of Bounded linear logic ([11]). A semantics for SLL formulas has been proposed in [8] and some expressiveness properties have been studied in [19].

For each of these systems one shows that the terms of the specific calculus can be evaluated in polynomial time. A completeness result is then proved by simulating in the calculus a standard model for PTIME computation such as PTIME Turing machines. It follows that all PTIME *functions* are representable in the calculus, which establishes its expressiveness. This does not mean that all *algorithms* are directly representable. For instance it has been observed that some common algorithms such as insertion sort or quicksort cannot be programmed in a natural way in the Bellantoni-Cook system (see for instance [12]). Important contributions to the study of programming aspects of Implicit computational complexity have been done in particular by Jones ([14]), Hofmann ([12]) and Marion ([18]).

In the present work we investigate the ideas underlying SLL and their application to programming. In [15] SLL is defined with sequent-calculus and the results are proved using proof-nets, a graph representation of proofs. In order to facilitate the study of programming we define a specific calculus, Soft lambda-calculus (SLC) which can be typed in Soft linear (or affine) logic, thus providing a term syntax for this logic. We show that the untyped version of this calculus satisfies the property of *polynomial strong normalization*: given a term, the length of any reduction sequence is bounded by a polynomial of its size. This generalizes the property of polynomial strong normalization of SLL from [15] (actually it was already pointed out by Lafont that the result would apply to untyped proof-nets). Our calculus is inspired from Terui's Light affine lambda-calculus ([20]) which is a calculus typable in Light affine logic and with polynomial strong normalization.

As untyped SLC already enjoys polynomial reduction we can then consider more liberal type systems allowing for more programming facilities. We propose a type system extending Soft affine logic with recursive types. We finally examine how this system enables to define new datatypes which might allow representing more algorithms. We illustrate our approach on the example of lists and the insertion sort algorithm.

Acknowledgements. We wish to thank Marcel Masseron for the stimulating discussions we had together on Soft linear logic and which led to the present paper. We are grateful to Kazushige Terui for his many comments and suggestions. Finally we thank the anonymous referees who helped improving the paper by several useful remarks.

2 Soft lambda-calculus

The introduction of our calculus will be done in two steps (as in [20]): first we will define a grammar of pseudo-terms and then we will distinguish terms among pseudo-terms. The *pseudo-terms* of Soft lambda-calculus (SLC) are defined by the grammar:

$$t, t' ::= x \mid \lambda x.t \mid (t\ t') \mid !t \mid \text{let } t \text{ be } !x \text{ in } t'$$

Given a pseudo-term t we denote by $FV(t)$ its set of free variables and for a variable x by $no(x,t)$ the number of free occurrences of x in t. A pseudo-term of the form $\text{let } u \text{ be } !x \text{ in } t_1$ is called a *let expression* and the variable x in it is bound:

$$FV(\text{let } u \text{ be } !x \text{ in } t_1) = FV(u) \cup FV(t_1) \backslash \{x\}\,.$$

If $\overrightarrow{t}$ and $\overrightarrow{x}$ respectively denote finite sequences of same length $(t_1, \ldots, t_n)$ and $(x_1, \ldots, x_n)$, then $\text{let } \overrightarrow{t} \text{ be } !\overrightarrow{x} \text{ in } t'$ will be an abbreviation for n consecutive *let* expressions on t_is and x_is: $\text{let } t_1 \text{ be } !x_1 \text{ in let } t_2 \text{ be } !x_2 \text{ in } \ldots t'$. In the case where $n = 0$, $\text{let } \overrightarrow{t} \text{ be } !\overrightarrow{x} \text{ in } t'$ is t'.

We define the *size* $|t|$ of a pseudo-term t by:

$$|x| = 1, \qquad |\lambda x.t| = |t| + 1, \qquad |(t_1\ t_2)| = |t_1| + |t_2|,$$
$$|!t| = |t| + 1, \quad |\text{let } t_1 \text{ be } !x \text{ in } t_2| = |t_1| + |t_2| + 1.$$

We will type these pseudo-terms in intuitionistic soft *affine* logic (ISAL). The formulas are given by the following grammar:

$$T ::= \alpha \mid T \multimap T \mid \forall \alpha.T \mid !\,T$$

We choose the affine variant of Soft linear logic, which means permitting full weakening, to allow for more programming facility. This does not change the polytime nature of the system, as was already the case for light logic ([3]).

We give the typing rules in a sequent calculus presentation. It offers the advantage of being closer to the logic. It is not so convenient for type-inference, but it is not our purpose in this paper. The typing rules are given in Figure 1.

For (right $\forall$) we have the condition: (*) α does not appear free in Γ.

Observe that the *let* expression is used to interpret both the *multiplexing* (mplex) and the *promotion* (prom.) logical rules. We could distinguish two different kinds of *let* but we prefer to have a small calculus.

For instance one can consider for unary integers the usual type of Linear logic: $N = \forall \alpha.!(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$. The integer n is represented by the following pseudo-term of type N, with n occurrences of s':

$$\lambda s.\lambda x.\text{let } s \text{ be } !s' \text{ in } (s'\ (s'\ (s' \ldots x) \ldots)$$

Among pseudo-terms we define a subclass of *terms*. These will be defined inductively together with a notion of *temporary variables*. The temporary variables of a term t, $TV(t)$, will be part of the free variables of t: $TV(t) \subseteq FV(t)$.

Definition 1. *The set $\mathcal{T}$ of terms is the smallest subset of pseudo-terms such that:*

$$\frac{}{x : A \vdash x : A}\ \text{(variable)} \qquad \frac{\Gamma \vdash t : A \quad \Delta, x : A \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}\ \text{(cut)}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \text{(right arrow)} \qquad \frac{\Gamma, x : B \vdash t : C \quad \Delta \vdash u : A}{\Gamma, \Delta, y : A \multimap B \vdash t[(yu)/x] : C}\ \text{(left arrow)}$$

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ \text{(weak.)} \qquad \frac{x_1 : A, \ldots, x_n : A, \Gamma \vdash t : B}{y : !A, \Gamma \vdash \text{let } y \text{ be } !x \text{ in } t[x/x_1, \ldots, x_n] : B}\ \text{(mplex)}$$

$$\frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : B}{y_1 : !A_1, \ldots, y_n : !A_n \vdash \text{let } \overrightarrow{y} \text{ be } !\overrightarrow{x} \text{ in } !t : !B}\ \text{(prom.)}$$

$$\frac{x : A[C/\alpha], \Gamma \vdash t : B}{x : \forall \alpha.A, \Gamma \vdash t : B}\ \text{(left } \forall\text{)} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall \alpha.B}\ \text{(right } \forall\text{) (*)}$$

Fig. 1. ISAL typing rules

1. $x \in \mathcal{T}$; *then* $TV(x) = \emptyset$;
2. $\lambda x.t \in \mathcal{T}$ *iff:* $x \notin TV(t)$, $t \in \mathcal{T}$ *and* $no(x,t) \leqslant 1$;
   then $TV(\lambda x.t) = TV(t)$;
3. $(t_1\ t_2) \in \mathcal{T}$ *iff:* $t_1, t_2 \in \mathcal{T}$, $TV(t_1) \cap FV(t_2) = \emptyset$, $FV(t_1) \cap TV(t_2) = \emptyset$;
   then $TV((t_1\ t_2)) = TV(t_1) \cup TV(t_2)$;
4. $!t \in \mathcal{T}$ *iff:* $t \in \mathcal{T}$, $TV(t) = \emptyset$ *and* $\forall x \in FV(t), no(x,t) = 1$;
   then $TV(!t) = FV(t)$;
5. $let\, t_1\ be\, !x\ in\, t_2 \in \mathcal{T}$ *iff:* $t_1, t_2 \in \mathcal{T}$, $TV(t_1) \cap FV(t_2) = \emptyset$, $FV(t_1) \cap TV(t_2) = \emptyset$;
   then $TV(let\, t_1\ be\, !x\ in\, t_2) = TV(t_1) \cup (TV(t_2) \backslash \{x\})$.

Basically the ideas behind the definition of terms are that:

- one can abstract only on a variable that is not temporary and which has at most one occurrence,
- one can apply ! to a term which has no temporary variable and whose free variables have at most one occurrence; the variables then become temporary;
- the only way to get rid of a temporary variable is to bind it using a *let* expression.

It follows from the definition that temporary variables in a term are linear:

Lemma 1. *If t is a term and $x \in TV(t)$, then $no(x,t) = 1$.*

The definition of depth will be useful later when discussing reduction:

Definition 2. *Let t be a term and u be an* occurrence *of subterm of t. We call* depth *of u in t, $d(u,t)$ the number d of subterms v of t such that u is a subterm of v and v is of the form $!v'$. The depth $d(t)$ of a term t is the maximum of $d(u,t)$ for u subterms of t.*

For instance: if $t = !(\lambda f.\lambda x.\text{let } f \text{ be } !f' \text{ in } !u)$ and $u = (f'x)$, we have $d(u,t) = 2$.

We can then observe that:

Proposition 1. *Let t be a term. If x belongs to $FV(t)$ and x_0 denotes an occurrence of x in t, then $d(x_0, t) \leqslant 1$. Moreover all occurrences of x in t have the same depth, that we can therefore denote by $d(x, t)$, and we have: $d(x, t) = 1$ iff $x \in TV(t)$.*

We will consider a subclass of terms:

Definition 3. *A term t is* well-formed *if:*
$TV(t) = \emptyset$ *and* $\forall x \in FV(t),\ no(x, t) = 1$.

Note that to transform an arbitrary term into a well-formed one, one only needs to add enough *let* expressions. Actually the properties we will prove in section 3 are valid for terms and the notion of well-formed terms is introduced only because these are the terms that can be duplicated during reduction. We have the following properties on terms and substitution:

Lemma 2. *If t is a term and $t = !t_1$, then t_1 is a well-formed term.*

Lemma 3. *If we have: (i) t, u terms, (ii) $TV(u) = \emptyset$, (iii) $x \notin TV(t)$, and (iv) $FV(u) \cap TV(t) = \emptyset$, then: $t[u/x]$ is a term and $TV(t[u/x]) = TV(t)$.*

We can then check the following:

Proposition 2. *If t is a pseudo-term such that in ISAL we have $\Gamma \vdash t : A$, then t is a well-formed term.*

Proof. We prove by induction on the ISAL derivation $\mathcal{D}$ the following statement:
i.h.($\mathcal{D}$): if the conclusion of $\mathcal{D}$ is $\Gamma \vdash t : A$ then: t is a term, $TV(t) = \emptyset$ and $\forall x \in \Gamma,\ no(x, t) \leqslant 1$.

All the cases of the induction follow directly from the application of definition 1 except (cut), (left arrow), (mplex) for which we also use lemma 3.

However not all well-formed terms are typable in ISAL: $t = \lambda x.\mathrm{let}\, x\, \mathrm{be}\, !y\, \mathrm{in}\, (y\ y)$ for instance is a well-formed term, but is not ISAL typable.

We will also need in the sequel two variants of lemma 3:

Lemma 4. *If we have: (i) t, u terms, (ii) $x \notin TV(t)$, (iii) $no(x, t) = 1$,*
(iv) $FV(u) \cap TV(t) = \emptyset$, (v) $TV(u) \cap FV(t) = \emptyset$, then: $t[u/x]$ is a term and $TV(t[u/x]) = TV(t) \cup TV(u)$.

Note that the main difference with lemma 3 is that we have here the assumption $no(x, t) = 1$.

Lemma 5. *If we have: (i) t is a term and u is a well-formed term, (ii) $x \in TV(t)$, (iii) $FV(u) \cap FV(t) = \emptyset$, then: $t[u/x]$ is a term and $TV(t[u/x]) = TV(t) \backslash \{x\} \cup FV(u)$.*

We now consider the contextual one-step reduction relation $\rightarrow^1$ defined on pseudo-terms by the rules of Figure 2. These rules assume renaming of bound variables so that capture of free variables is avoided in the usual way. The rules (com1) and (com2) are the commutation rules. The relation $\rightarrow$ is the transitive closure of $\rightarrow^1$.

We have:

(β):	$((\lambda x.t)\ u) \to^1 t[u/x]$
(!) :	$\mathrm{let}\,!u\,\mathrm{be}\,!x\,\mathrm{in}\,t \to^1 t[u/x]$
(com1):	$\mathrm{let}\,(\mathrm{let}\,t_1\,\mathrm{be}\,!y\,\mathrm{in}\,t_2)\,\mathrm{be}\,!x\,\mathrm{in}\,t_3 \to^1 \mathrm{let}\,t_1\,\mathrm{be}\,!y\,\mathrm{in}\,(\mathrm{let}\,t_2\,\mathrm{be}\,!x\,\mathrm{in}\,t_3)$
(com2):	$(\mathrm{let}\,t_1\,\mathrm{be}\,!x\,\mathrm{in}\,t_2)t_3 \to^1 \mathrm{let}\,t_1\,\mathrm{be}\,!x\,\mathrm{in}\,(t_2\ t_3)$

Fig. 2. reduction rules

Proposition 3. *The reduction is well defined on terms: if t is a term and $t \to^1 t'$ then t' is a term. Moreover:*

- *$FV(t') \subseteq FV(t)$ and $TV(t') \subseteq TV(t)$,*
- *if t is well-formed then t' is well-formed.*

Proposition 4 (local confluence). *The reduction relation $\to^1$ on terms is locally confluent: if $t \to^1 t'_1$ and $t \to^1 t'_2$ then there exists t' such that $t'_1 \to t'$ and $t'_2 \to t'$.*

3 Bounds on the reduction

We want to find a polynomial bound on the length of reduction sequences of terms, similar to that holding for SLL proof-nets ([15]). For that we must define a parameter on terms corresponding to the maximal arity of the multiplexing links in SLL proof-nets.

Definition 4. *The* rank *$rank(t)$ of a term t is defined inductively by:*

$$rank(x) = 0, \qquad rank(!t) = rank(t),$$
$$rank(\lambda x.t) = rank(t), \qquad rank((t_1\ t_2)) = \max(rank(t_1), rank(t_2)),$$
$$rank(let\,u\,be\,!x\,in\,t_1) = \begin{cases} \max(rank(u), rank(t_1)) & \text{if } x \in TV(t_1), \\ \max(rank(u), rank(t_1), no(x, t_1)) & \text{if } x \notin TV(t_1). \end{cases}$$

The first case in the definition of $rank(\mathrm{let}\,u\,\mathrm{be}\,!x\,\mathrm{in}\,t_1)$ corresponds to a promotion, while the second one corresponds to a multiplexing and is the key case in this definition.

To establish the bound we will adapt the argument given by Lafont for proof-nets. First we define for a term t and an integer n the *weight* $W(t, n)$ by:

$$W(x, n) = 1,$$
$$W(\lambda x.t, n) = W(t, n) + 1, \qquad W((t_1\ t_2), n) = W(t_1, n) + W(t_2, n),$$
$$W(!u, n) = nW(u, n) + 1, \qquad W(\mathrm{let}\,u\,\mathrm{be}\,!x\,\mathrm{in}\,t_1, n) = W(u, n) + W(t_1, n).$$

We have the following key lemma:

Lemma 6. *Let t be a term and $n \geqslant rank(t)$.*

1. *if $x \notin TV(t)$ and $no(x, t) = k$, then: $W(t[u/x], n) \leqslant W(t, n) + kW(u, n)$.*
2. *if $x \in TV(t)$ then: $W(t[u/x], n) \leqslant W(t, n) + nW(u, n)$.*

Proposition 5. *Let t be a term and $n > rank(t)$. If $t \rightarrow^1 t'$ by a (β) or (!) reduction rule then $W(t', n) < W(t, n)$.*

Proof. If $t \xrightarrow{\sigma} t'$ with $\sigma = (\beta)$ or (!) then there is a context C and a redex r such that $t = C[r]$, $t' = C[r']$ and $r \xrightarrow{\sigma} r'$.

We prove the statement by induction on the context C, for a given $n > rank(t)$. Let us consider the basic case of the empty context, *i.e.* $t = r$ using the definitions of terms and rank, and lemma 6:

for instance for a (!) reduction rule,

$r = \text{let } !u \text{ be } !x \text{ in } r_1, \qquad r' = r_1[u/x]$

$W(r, n) = W(\text{let } !u \text{ be } !x \text{ in } r_1, n) = n.W(u, n) + 1 + W(r_1, n)$.

If $x \in TV(r_1)$ then by lemma 6 $W(r', n) < W(r, n)$, otherwise we have $x \in FV(r_1) \backslash TV(r_1)$ and:

$$\begin{aligned} W(r', n) &\leqslant W(r_1, n) + no(x, r_1).W(u, n) \\ &\leqslant W(r_1, n) + rank(r).W(u, n) \\ &\leqslant W(r_1, n) + n.W(u, n) < W(r, n)\,. \end{aligned}$$

The case of a (β) reduction is easy.

The induction on C is straightforward, using in the case $C = !C_1$ the fact that $n \geqslant 1$ as we have the strict inequality $n > rank(t)$.

For the commutation rules we have $W(t', n) = W(t, n)$. So we need to use a measure of the commutations in a reduction sequence to be able to bound the global length. We make an adaptation of the weight used in [20].

Given an integer n and a term t, for each subterm occurrence in t of the form $t_1 = \text{let } u \text{ be } !x \text{ in } t_2$, we define the *measure* of t_1 in t by:

$$m(t_1, t) = W(t, n) - W(t_2, n)$$

and $M(t, n)$ the *measure* of t by the sum of $m(t_1, t)$ for all subterms t_1 of t which are *let* expressions.

Proposition 6. *Let t be a term and $n > rank(t)$. If $t \rightarrow^1 t'$ by a commutation reduction rule then $M(t', n) < M(t, n)$.*

Given a term t we denote by $nlet(t)$ the number of subterm occurrences of *let* expressions in t.

Lemma 7. *Let t be a term and $n \geqslant 1$. We have $nlet(t) \leqslant W(t, n) - 1$.*

Proposition 7. *If t is a term and $p = d(t)$, $k = W(t, 1)$, and $n \geqslant 1$ then:*

$$W(t, n) \leqslant k.n^p$$

Theorem 1. *[Polynomial strong normalization]*

For any integer d there is a polynomial P_d (with degree linear in d) such that:

for any term t of depth d, any sequence of reductions of t has length bounded by $P_d(|t|)$.

Proof. Let t be a term of depth d and $n > rank(t)$. We will call *round* a sequence of reductions and *proper round* a non empty sequence of (β) and (!) reductions.

If $t \xrightarrow{\sigma} t'$ then there is an integer l such that σ can be described by an alternate sequence of commutation rules rounds and proper rounds as follows:

$$t = t_1 \overset{(com)}{\rightarrow^\star} t_2 \overset{(\beta),(!)}{\rightarrow^\star} t_3 \ \ldots \ t_{2i+1} \overset{(com)}{\rightarrow^\star} t_{2i+2} \overset{(\beta),(!)}{\rightarrow^\star} t_{2i+3} \ \ldots \ t_{2l+1} \overset{(com)}{\rightarrow^\star} t_{2l+2} = t'$$

Remark that the alternate sequence starts and finishes with a commutation rules round. The sequence σ contains l proper rounds. Because each such round strictly decreases the weight of t (Prop.5) and the commutation rules leave the weight unchanged we have $l \leqslant W(t,n)$. Moreover the length of all proper rounds in σ is bounded by $W(t,n)$.

On the other hand we have by definition and lemma 7:

$$M(t',n) < nlet(t').W(t',n) \leqslant (W(t',n))^2 - W(t',n) \leqslant (W(t,n))^2 - W(t,n)\,.$$

There are at most $(l+1)$ commutation rules rounds, so by Prop. 6 the length of all such rounds is bounded by $(l+1).((W(t,n))^2 - W(t,n))$. Then we deduce:

$$|\sigma| \leqslant (l+1).((W(t,n))^2 - W(t,n)) + W(t,n) \leqslant (W(t,n))^3$$

Finally this result can be applied to any $n > rank(t)$. One can check that for any pseudo-term t we have $|t| > rank(t)$. Consider $n = |t|$, by Prop.7 we obtain that

$$|\sigma| \leqslant (W(t,1))^3.(|t|)^{3d} \leqslant |t|^{3(d+1)}$$

where $d = d(t)$.

Remark 1. If a term t of depth d corresponds to a program and u to an argument such that $d(u) \leqslant d$, then $(t\ u)$ normalizes in at most $Q_d(|u|)$ steps for some polynomial Q_d:

by the previous theorem if $(t\ u) \xrightarrow{\sigma} t'$ then $|\sigma| \leqslant (|t| + |u|)^{3(d+1)}$ because $d((t\ u)) = d(t) = d$. Let $Q_d(X)$ be the following polynomial :

$$Q_d(X) = (X + |t|)^{3(d+1)}.$$

Note that theorem 1 shows that the calculus is *strongly* polytime in the sense of [20]: there exists a polynomial bounding the length of *any* reduction sequence (no matter the reduction strategy). An obvious consequence is then:

Corollary 1 (Strong normalization). *The terms of soft lambda calculus are strongly normalizing.*

Corollary 2 (Confluence property). *If a term t is such that $t \rightarrow u$ and $t \rightarrow v$ then there exists a term w such that $u \rightarrow w$ and $v \rightarrow w$.*

Proof. By local confluence (Proposition 4) and strong normalization.

$$\frac{\Gamma, x_1 : A_1, x_2 : A_2 \vdash t : B}{\Gamma, x : A_1 \otimes A_2 \vdash \text{let } x \text{ be } x_1 \otimes x_2 \text{ in } t : B}\ (\text{left } \otimes) \qquad \frac{\Gamma_1 \vdash t_1 : A_1 \quad \Gamma_2 \vdash t_2 : A_2}{\Gamma_1, \Gamma_2 \vdash t_1 \otimes t_2 : A_1 \otimes A_2}\ (\text{right } \otimes)$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \ \text{inl}(t) : A \oplus B}\ (\text{right } \oplus_1) \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \ \text{inr}(t) : A \oplus B}\ (\text{right } \oplus_2)$$

$$\frac{\Gamma, x_1 : A_1 \vdash t_1 : B \quad \Gamma, x_2 : A_2 \vdash t_2 : B}{\Gamma, x : A_1 \oplus A_2 \vdash \text{case } x \text{ of inl}(x_1) \Rightarrow t_1 \mid \text{inr}(x_2) \Rightarrow t_2 \ : B}\ (\text{left } \oplus)$$

Fig. 3. Derived rules

4 Extension of the calculus

Thanks to full weakening, the connectives $\otimes$ and $\oplus$, (as well as & and $\exists$) and the constant 1 are definable from $\{\multimap, \forall\}$ ([3], [21]):

$$A \otimes B = \forall\alpha.((A \multimap B \multimap \alpha) \multimap \alpha) \qquad 1 = \forall\alpha.(\alpha \multimap \alpha)$$
$$A \oplus B = \forall\alpha.((A \multimap \alpha) \multimap (B \multimap \alpha) \multimap \alpha)$$

We use as syntactic sugar the following new constructions on terms with the typing rules of Figure 3 (we follow the presentation of [1]):

$(t_1 \otimes t_2)$, let u be $x_1 \otimes x_2$ in t,
$\text{inl}(t)$, $\text{inr}(t)$, case u of $\text{inl}(x) \Rightarrow t_1 \mid \text{inr}(y) \Rightarrow t_2$;

We denote by 1 the closed term of type 1. The derived reduction rules for these constructions are:

$\text{let } (t_1 \otimes t_2) \text{ be } x_1 \otimes x_2 \text{ in } u \ \rightarrow\ u[t_1/x_1, t_2/x_2]$
$\text{case } \text{inl}(u) \text{ of inl}(x_1) \Rightarrow t_1 \mid \text{inr}(x_2) \Rightarrow t_2 \ \rightarrow\ t_1[u/x_1]$
$\text{case } \text{inr}(u) \text{ of inl}(x_1) \Rightarrow t_1 \mid \text{inr}(x_2) \Rightarrow t_2 \ \rightarrow\ t_2[u/x_2]$

We also use as syntactic sugar, for x a variable: $\text{let } u \text{ be } x \text{ in } t \stackrel{def}{=} ((\lambda x.t)\ u)$. We now enlarge the language of types with a fixpoint construction:

$$T ::= \alpha \mid T \multimap T \mid \forall\alpha.T \mid !\,T \mid \mu\alpha.T$$

We add the corresponding typing rule and denote by ISALF, intuitionistic light affine logic with fixpoints, the new system: Figure 4.

Proposition 8. *If t is a pseudo-term typable in ISALF then t is a well-formed term.*

Proof. One simply extends the inductive proof of Prop. 2 to ISALF derivations. We have four new rules to consider but as the i.h. does not make any use of the types in the judgement these cases are all trivial.

Proposition 9 (Subject reduction). *If we have in the system ISALF $\Gamma \vdash t : A$ and $t \ \rightarrow\ t'$ then $\Gamma \vdash t' : A$.*

Basically this result follows from the fact that as a logical system ISALF admits cut-elimination.

the typing rules of ISAL and

$$\frac{x : \mu X.A, \Gamma \vdash t : B}{x : A[\mu X.A/X], \Gamma \vdash t : B}\ \text{(left unfold)} \qquad \frac{\Gamma \vdash t : \mu X.A}{\Gamma \vdash t : A[\mu X.A/X]}\ \text{(right unfold)}$$

$$\frac{x : A[\mu X.A/X], \Gamma \vdash t : B}{x : \mu X.A, \Gamma \vdash t : B}\ \text{(left fold)} \qquad \frac{\Gamma \vdash t : A[\mu X.A/X]}{\Gamma \vdash t : \mu X.A}\ \text{(right fold)}$$

Fig. 4. ISALF typing rules

Note that even though we have no restriction on the types on which we take fixpoints, the typed terms are always normalizable and have a polynomial bound on the length of their reduction. This follows from the fact that the polynomial termination result (Theorem 1) already holds for untyped terms.

In the following we will handle terms typed in ISALF. Rather than giving the explicit type derivations in the previous system, which is a bit tedious because it is a sequent-calculus style presentation, we will use a Church typing notation. The recursive typing rules and second-order rules will be left implicit. From this notation it is possible to reconstruct an explicit type derivation if needed. Here is an example of typed term (integer 2 in unary representation)

$$\lambda s^{!(\alpha \multimap \alpha)}.\lambda x^{\alpha}.\text{let } s \text{ be } !s' \text{ in } (s'\ (s'\ x))^{\alpha}\ : N\,.$$

5 Datatypes and list processing

5.1 Datatypes for lists

Given a type A, we consider the following types defining lists of elements of A:

$$\mathcal{L}(A) = \forall\alpha.!(A \multimap \alpha \multimap \alpha) \multimap \alpha \multimap \alpha, \qquad L(A) = \mu X.(1 \oplus (A \otimes X)).$$

The type $\mathcal{L}(A)$ is the adaptation of the usual system F type for lists. It supports an iteration scheme, but does not enable to define in SLC a *cons* function with type $\mathcal{L}(A) \multimap A \multimap \mathcal{L}(A)$. This is analog to the fact that N does not allow for a successor function with type $N \multimap N$ ([15]).

The type $L(A)$ on the contrary allows to define the usual basic functions on lists *cons*, *tail*, *head*, but does not support iteration. The empty list for type $L(A)$ is given by $\epsilon = \text{ inl}(1)$ and the basic functions by:

$$\begin{aligned}
\mathit{cons} &= \lambda l^{L(A)}.\lambda a^{A}.\ \text{inr}(a \otimes l)\ :\ L(A) \multimap A \multimap L(A)\\
\mathit{tail} &= \lambda l^{L(A)}.\text{case } l \text{ of inl}(l') \Rightarrow\ \text{inl}(l')\\
&\qquad\qquad\qquad\quad \mid \text{inr}(l') \Rightarrow \text{let } l' \text{ be } a \otimes l'' \text{ in } l''\ :\ L(A) \multimap L(A)\\
\mathit{head} &= \lambda l^{L(A)}.\text{case } l \text{ of inl}(l') \Rightarrow\ a_0\\
&\qquad\qquad\qquad\quad \mid \text{inr}(l') \Rightarrow \text{let } l' \text{ be } a \otimes l'' \text{ in } a\ :\ L(A) \multimap A
\end{aligned}$$

where a_0 is a dummy value returned by *head* if the list is empty. We would like to somehow bring together the advantages of $\mathcal{L}(A)$ and $L(A)$ in a single datatype. This is what we will try to do in the next sections.

5.2 Types with integer

Our idea is given a datatype A to add to it a type N so as to be able to iterate on A. The type $N \otimes A$ would be a natural candidate, but it does not allow for a suitable iteration. We therefore consider the following type:

$$N[A] = \forall\alpha.!(\alpha \multimap \alpha) \multimap \alpha \multimap (A \otimes \alpha)$$

Given an integer n and a closed term a of type A, we define an element of $N[A]$:

$$n[a] = \lambda s^{!(\alpha\multimap\alpha)}.\lambda x^{\alpha}.(a^{A} \otimes \mathrm{let}\, s \,\mathrm{be}\, !s' \,\mathrm{in}\, (s'\ s'\ldots\ s'x)^{\alpha}) \ : N[A]$$

where s' is repeated n times.

We can give terms allowing to extract from an element $n[a]$ of type $N[A]$ either the data a or the integer n.

$$extractd : N[A] \multimap A \qquad extractint : N[A] \multimap N$$

For instance:

$extractd = \lambda p^{N[A]}.\mathrm{let}\,(p\ !id^{\beta\multimap\beta}\ id^{\alpha\multimap\alpha}) \,\mathrm{be}\, a^{A} \otimes r^{\alpha} \,\mathrm{in}\, a\,,$

where id is the identity term and $\beta = \alpha \multimap \alpha$.

However it does not seem possible to extract both the data and the integer with a term of type $N[A] \multimap N \otimes A$. On the contrary from n and a one can build $n[a]$ of type $N[A]$:

$build = \lambda t.\mathrm{let}\, t \,\mathrm{be}\, n \otimes a \,\mathrm{in}\, \lambda s.\lambda x.(a \otimes (n\ s\ x)) \ : N \otimes A \multimap N[A]\,.$

We can turn the construction $N[.]$ into a functor: let us define the action of $N[.]$ on a closed term $f : A \multimap B$ by

$$N[f] = \lambda p^{N[A]}.\lambda s^{!(\alpha\multimap\alpha)}.\lambda x^{\alpha}.\mathrm{let}\,(p\ s\ x)^{A\otimes\alpha} \,\mathrm{be}\, a \otimes r \,\mathrm{in}\, ((f\ a)^{B} \otimes r^{\alpha})\,.$$

Then $N[f] : N[A] \multimap N[B]$, and $N[.]$ is a functor.

We have the following principles:

$$absorb : N[A] \otimes B \multimap N[A \otimes B]\,, \qquad out : N[A \multimap B] \multimap (A \multimap N[B])\,.$$

The term $absorb$ for instance is defined by:

$absorb = \lambda t^{N[A]\otimes B}.\lambda s^{!(\alpha\multimap\alpha)}.\lambda x^{\alpha}.\mathrm{let}\, t \,\mathrm{be}\, p \otimes b \,\mathrm{in}$
$\mathrm{let}\,(p\ s\ x) \,\mathrm{be}\, a \otimes r \,\mathrm{in}\, (a \otimes b \otimes r)^{A\otimes B\otimes\alpha}\,.$

5.3 Application to lists

In the following we will focus our interest on lists. We will use as a shorthand notation $L'(A)$ for $N[L(A)]$. The terms described in the previous section can be specialized to this particular case.

In practice here we will use the type $L'(A)$ with the following meaning: the elements $n[l]$ of $L'(A)$ handled are expected to be such that the list l has a length inferior or equal to n. We will then be able to do iterations on a list up to its length.

The function *erase* maps $n[l]$ to $n[\epsilon]$ where ϵ is the empty list; it is obtained by a small modification on *extractint*:

$$\begin{aligned} &erase \ : \ L'(A) \multimap L'(A) \\ &erase = \lambda p^{L'(A)}.\lambda s^{!(\alpha \multimap \alpha)}.\lambda x^{\alpha}.\mathrm{let}\,(p\ s\ x)\,\mathrm{be}\,l^{L(A)} \otimes r^{\alpha}\,\mathrm{in}\,(\epsilon^{L(A)} \otimes r^{\alpha}) \end{aligned}$$

We have for the type $L'(A)$ an iterator given by:

$$\begin{aligned} &Iter \ : \ \forall\alpha.!(\alpha \multimap \alpha) \multimap \alpha \multimap L'(A) \multimap (L(A) \otimes \alpha) \\ &Iter = \lambda F^{!(\alpha \multimap \alpha)}.\lambda e^{\alpha}.\lambda l^{L'(A)}.(l\ F\ e) \end{aligned}$$

If F has type $B \multimap B$, e type B and F has free variables $\overrightarrow{x}$ then if $f = (Iter\ (\mathrm{let}\ \overrightarrow{y}\ \mathrm{be}\,!\overrightarrow{x}\,\mathrm{in}\,!F)\ e)$ we have:

$$(f\ n[l]) \ \rightarrow \ l \otimes (\mathrm{let}\ \overrightarrow{y}\ \mathrm{be}\,!\overrightarrow{x}\,\mathrm{in}\,(F \ldots (F\ e)\ldots),$$

where in the r.h.s. term F is repeated n times. Such an iterator can be in fact described more generally for any type $N[A]$ instead of $N[L(A)]$.

Using iteration we can build a function which reconstructs an element of $L'(A)$; it acts as an identity function on $L'(A)$ but is interesting though because in the sequel we will need to *consume and restore* integers in this way:

$$\begin{aligned} F = \mathrm{let}\,s\,\mathrm{be}\,!s'^{\alpha \multimap \alpha}\,\mathrm{in}\,!(\lambda r^{\alpha}.(s'r)^{\alpha}) \ : !(\alpha \multimap \alpha), \ \text{with } FV(F) = \{s^{!(\alpha \multimap \alpha)}\} \\ reconstr = \lambda p^{L'(A)}.\lambda s^{!(\alpha \multimap \alpha)}.\lambda x^{\alpha}.(Iter\ F\ x\ p) \ : L'(A) \multimap L'(A) \end{aligned}$$

Given terms $t : A \multimap B$ and $u : B \multimap C$ we will denote by $t; u : A \multimap C$ the *composition* of t and u defined as $(\lambda a^{A}.(u\ (t\ a)))$.

Finally we have the usual functions on lists with type $L'(A)$, using the ones defined before for the type $L(A)$:

$$\begin{aligned} tail' &= N[tail] &&: L'(A) \multimap L'(A) \\ head' &= N[head]; extractd &&: L'(A) \multimap A \\ cons' &= N[cons]; out &&: L'(A) \multimap A \multimap L'(A) \end{aligned}$$

Note that to preserve the invariant on elements of $L'(A)$ mentioned at the beginning of the section we will need to apply $cons'$ to elements $n[l]$ such that $n \geqslant m + 1$ where m is the length of l.

5.4 Example: insertion sort

We illustrate the use of the type $N[L(A)]$ by giving the example of the insertion sort algorithm. Contrarily to the setting of Light affine logic with system F like types, we can here define functions obtained by successive nested structural recursions. Insertion sort provides such an example with two recursions. We use the presentation of this algorithm described in [13].

The type A represents a totally ordered set (we denote the order by $\leqslant$) that we suppose to be finite for simplification. Let us assume that we have for A a comparison function which returns its inputs:

$$comp : A \otimes A \multimap A \otimes A, \quad \text{with } (comp\ a_0\ a_1) \rightarrow \begin{cases} (a_0 \otimes a_1) & \text{if } a_0 \leqslant a_1\,, \\ (a_1 \otimes a_0) & \text{otherwise.} \end{cases}$$

The function *comp* can in fact be defined in SLC.

Insertion in a sorted list.

Let a_0 be an arbitrary element of type A. We will do an iteration on type: $B = L(A) \multimap A \multimap L(A) \otimes \alpha$. The iterated function will reconstruct the integer used for its iteration. Let us take $F : !(B \multimap B)$ with $FV(F) = \{s^{!(\alpha \multimap \alpha)}\}$, given by:

$$\begin{array}{ll} F = & \text{let } s \text{ be } !s'^{\alpha \multimap \alpha} \text{ in} \\ & !(\lambda\phi^B.\lambda l^{L(A)}.\lambda a^A. \\ & \qquad \text{case } l \text{ of } \ \text{inl}(l_1) \ \Rightarrow\ \text{let } (\phi\ \epsilon\ a_0) \text{ be } l' \otimes r^\alpha \text{ in} \\ & \qquad\qquad (cons\ a\ \epsilon)^{L(A)} \otimes (s'\ r)^\alpha \\ & \qquad\quad \mid \text{inr}(l_1) \ \Rightarrow\ \text{let } l_1 \text{ be } b \otimes l' \text{ in} \\ & \qquad\qquad \text{let } (comp\ a\ b) \text{ be } a_1 \otimes a_2 \text{ in} \\ & \qquad\qquad\quad \text{let } (\phi\ l'\ a_2) \text{ be } l'' \otimes r \text{ in} \\ & \qquad\qquad\qquad (cons\ a_1\ l'') \otimes (s'\ r)^\alpha) \end{array}$$

Let $e : B$ be the term $e = \lambda l^{L(A)}.\lambda a^A.(\epsilon^{L(A)} \otimes x^\alpha)$. Note that $FV(e) = \{x^\alpha\}$. Then we have: $s : !\alpha \multimap \alpha, x : \alpha \vdash (Iter\ F\ e) : L'(A) \multimap L(A) \otimes B$.

Finally we define:

$$\begin{array}{l} insert = \lambda p^{L'(A)}.\lambda a^A.\ \lambda s^{!(\alpha \multimap \alpha)}.\lambda x^\alpha. \\ \qquad\qquad \text{let } (Iter\ F\ e\ p)^{L(A) \otimes B} \text{ be } l^{L(A)} \otimes f^B \text{ in } (f\ l\ a)^{L(A) \otimes \alpha} \end{array}$$

and get: $insert : L'(A) \multimap A \multimap L'(A)$.

Insertion sort.

We define our sorting program by iteration on $B = L(A) \otimes L'(A)$. The left-hand-side list is the list to process while the r.h.s. one is the resulting sorted list. Then $F : !(B \multimap B)$ is the closed term given by:

$$\begin{array}{l} F = !(\lambda t^B.\text{let } t \text{ be } l_1^{L(A)} \otimes p^{L'(A)} \text{ in case } l_1 \text{ of} \\ \qquad\qquad \text{inl}(l_2) \ \Rightarrow\ \text{inl}\,(l_2) \otimes p \\ \qquad\qquad \mid \text{inr}(l_2) \ \Rightarrow\ \text{let } l_2 \text{ be } a \otimes l_3 \text{ in } (l_3^{L(A)} \otimes (insert\ p\ a)^{L'(A)}\ ) \end{array}$$

$e = l^{L(A)} \otimes (erase\ p_0)^{L'(A)} : B$

We then have:

$l : L(A), p_0 : L'(A) \vdash (Iter\ F\ e) : L'(A) \multimap L(A) \otimes B$

So we define:

$$\begin{array}{l} presort = \lambda p_0^{L'(A)}.\lambda p_1^{L'(A)}.\lambda p_2^{L'(A)}. \\ \qquad\qquad \text{let } (extractd\ p_1) \text{ be } l^{L(A)} \text{ in} \\ \qquad\qquad\quad \text{let } (Iter\ F\ e\ p_2) \text{ be } l' \otimes l'' \otimes p' \text{ in } p' \end{array}$$

Using multiplexing we then get:

$$sort = \lambda p^{!L'(A)}.\text{let } p \text{ be } !p'^{L'(A)} \text{ in } (presort\ p'\ p'\ p')^{L'(A)}\ :\ !L'(A) \multimap L'(A)$$

Remark 2. More generally the construction $N[.]$ can be applied successively to define the following family of types: $N^{(0)}[A] = A,\ N^{(i+1)}[A] = N[N^{(i)}[A]]$.

This allows to type programs obtained by several nested structural recursions. For instance insertion sort could be programmed with a type of the form $N^{(2)}[A] \multimap N^{(2)}[A]$. This will be detailed in a future work.

5.5 Iteration

We saw that with the previous iterator *Iter* one could define from $F : B \multimap B$ and $e : B$ an f such that: $(f\ \ l[n]) \ \ \rightarrow \ \ l \otimes (\text{let } \overrightarrow{y} \text{ be } !\overrightarrow{x} \text{ in } (F \ldots (F\ \ e) \ldots))$. However the drawback here is that l is not used in e. We can define a new iterator which does not have this default, using the technique already illustrated by the *insertion* term. Given a type variable α, we set $C = L(A) \multimap \alpha$.

If g is a variable of type $!(\alpha \multimap \alpha)$, we define:

$$G' = \text{let } g^{!(\alpha \multimap \alpha)} \text{ be } !g' \text{ in } !(\lambda b'^{C}.\lambda l^{L(A)}.(g'\ (b'\ l)))^{\alpha} : !(C \multimap C)$$

Then: $It = \lambda g^{!(\alpha \multimap \alpha)}.\lambda e^{C}.\lambda p^{L'(A)}.\text{let } (Iter\ G'\ e^{C}\ p) \text{ be } l_1^{L(A)} \otimes f^{C} \text{ in } (f\ l_1)^{\alpha}$
$It : \ \forall\alpha.!(\alpha \multimap \alpha) \multimap (L(A) \multimap \alpha) \multimap L'(A) \multimap \alpha$

So if $f = (It\ (\text{let } \overrightarrow{y} \text{ be } !\overrightarrow{x} \text{ in } !F)\ \lambda l_0.e')$ we have:

$$(f\ l[n]) \ \rightarrow \ \text{let } \overrightarrow{y} \text{ be } !\overrightarrow{x} \text{ in } (F \ldots (F\ e'[l/l_0]) \ldots),$$

where in the r.h.s. term F is repeated n times.

6 Conclusion and future work

We studied a variant of lambda-calculus (SLC) which can be typed in Soft affine logic and is intrinsically polynomial. The contribution of the paper is twofold:

(i) We showed that the ideas at work in Soft linear logic to control duplication can be used in a lambda-calculus setting with a concise untyped language. Note that the language of our calculus is simpler than those of calculi corresponding to ordinary linear logic such as in [5, 1]. Even if the underlying intuitions come from proof-nets and Lafont's results, we think that this new presentation will facilitate further study of Soft logic.

(ii) We investigated the use of recursive types in conjunction with Soft logic. They allowed us to define non-standard types for lists and we illustrated the expressiveness of SLC by programming the insertion sort algorithm.

We think SLC provides a good framework to study the algorithmic possibilities offered by the ideas of Soft logic. One drawback of the examples we gave here is that their programming is somehow too low-level. One would like to have some generic way of programming functions defined by structural recursion (with some conditions) that could be compiled into SLC. Current work in this direction is under way with Kazushige Terui. It would be interesting to be able to state sufficient conditions on algorithms, maybe related to space usage, for being programmable in SLC.

References

1. S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. A. Asperti and L. Roversi. Intuitionistic light affine logic (proof-nets, normalization complexity, expressive power). *ACM Transactions on Computational Logic*, 3(1):1–39, 2002.
3. Andrea Asperti. Light affine logic. In *Proceedings LICS'98*. IEEE Computer Society, 1998.
4. P. Baillot. Checking polynomial time complexity with types. In *Proceedings of International IFIP Conference on Theoretical Computer Science 2002*, Montreal, 2002. Kluwer Academic Press.
5. P.N. Benton, G.M. Bierman, V.C.V. de Paiva, and J.M.E. Hyland. A term calculus for intuitionistic linear logic. In *Proceedings TLCA'93*, volume 664 of *LNCS*. Springer Verlag, 1993.
6. S. Bellantoni and S. Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
7. S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3), 2000.
8. U. Dal Lago and S. Martini. Phase semantics and decidability results for elementary and soft linear logics. submitted, 2003.
9. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
10. J.-Y. Girard. Light linear logic. *Information and Computation*, 143:175–204, 1998.
11. J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science*, 97:1–66, 1992.
12. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings LICS'99*. IEEE Computer Society, 1999.
13. M. Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3), 2000.
14. N. Jones. *Computability and complexity, from a programming perspective*. MIT Press, 1997.
15. Y. Lafont. Soft linear logic and polynomial time. to appear in Theoretical Computer Science, 2004.
16. D. Leivant. Predicative recurrence and computational complexity I: word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhauser, 1994.
17. D. Leivant and J.-Y. Marion. Lambda-calculus characterisations of polytime. *Fundamenta Informaticae*, 19:167–184, 1993.
18. J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique*. PhD thesis, Université de Nancy, 2000. Habilitation à diriger les recherches.
19. H. Mairson and K. Terui. On the computational complexity of cut-elimination in Linear logic. In *Proceedings of ICTCS 2003*, volume 2841 of *LNCS*, pages 23–36. Springer, 2003.
20. K. Terui. Light Affine Lambda-calculus and polytime strong normalization. In *Proceedings LICS'01*. IEEE Computer Society, 2001.
21. K. Terui. *Light Logic and Polynomial Time Computation*. PhD thesis, Keio University, 2002.