An Embedding of the BSS Model of Computation in Light Affine Lambda-Calculus.

Patrick Baillot, Marco Pedicini

An Embedding of the BSS Model of Computation in Light Affine Lambda-Calculus *

Patrick Baillot
LIPN, CNRS / Université Paris 13, France
patrick.baillot@lipn.univ-paris13.fr

Marco Pedicini
IAC, Consiglio Nazionale delle Ricerche, Roma, Italy
marco@iac.rm.cnr.it

Abstract

This paper brings together two lines of research: implicit characterization of complexity classes by Linear Logic (LL) on the one hand, and computation over an arbitrary ring in the Blum-Shub-Smale (BSS) model on the other. Given a fixed ring structure K we define an extension of Terui's light affine lambda-calculus typed in LAL (Light Affine Logic) with a basic type for K. We show that this calculus captures the polynomial time function class FP(K): every typed term can be evaluated in polynomial time and conversely every polynomial time BSS machine over K can be simulated in this calculus.

1 Introduction

BSS computation. The Blum-Shub-Smale (BSS) model was introduced as an extension of the classical model of Turing machines to describe computations on an arbitrary ring or field K ([BSS89]; see also [BCSS98]). The idea is basically to consider an idealized machine which can store elements of the ring and perform on them a certain number of operations or tests at unary cost. The initial interest was on computation over reals and one motivation was to get a framework to reason about complexity of algorithms from numerical analysis and applied mathematics. Complexity classes analogous to the ones of the classical setting have been defined and this setting subsumes the classical one in the case K is taken to be $\mathbb{Z}/2\mathbb{Z}$ (with expected boolean operations). Moreover this approach was later extended to arbitrary logical structures ([Poi95]).

One might object that the BSS model is questionable from the point of view of physical realization: performing equality tests on real numbers at unary cost for instance is problematic. Anyway observe that it provides a setting which is compatible with the common way of handling complexity in numerical analysis an symbolic computation; thus we think it is a relevant model.

Implicit computational complexity. In the classical computational paradigm some work has been done to characterize functions of various complexity classes without reference to a machine model and explicit resource bounds. This line of research, Implicit Computational Complexity (ICC), has been developed using various approaches such as recursion theory ([BC92, Lei94]), lambda-calculus ([LM93]) or logic ([Gir98]). On the practical side it has yielded techniques for automatically or partially automatically inferring complexity bounds on programs ([Hof99, Jon01, MM00]).

*Work partially supported by Projects "Interaction and Complexity" (cooperation project 2004-2006: CNR, Italy - CNRS, France), GEOCAL (ACI), NO-COST (ANR).

Linear logic (LL, [Gir87]) has provided one line of research in ICC which fits in the proofs-as-programs paradigm: variants of LL with strict resource duplication disciplines such as Light Linear Logic ([Gir98], or its variant Light Affine Logic, [Asp98]) or Soft Linear Logic ([Laf04]) capture deterministic polytime computation. Light Affine Logic (LAL) has in particular been studied using specific term calculi ([Asp98, Rov00]); among these, Terui's *light affine lambda-calculus* enjoys good properties and has allowed to prove new properties on LAL (like the strong polytime bound, see [Ter01]). Some advantages of the light logics approach are the fact that it allows higher-order computation (also the case in [Hof00, BNS00]), polymorphism and enables to define new datatypes (as in system F). It fits also well with program extraction from termination proofs: in [Gir98, Ter04]) a naive set theory is presented in which the provably total functions are exactly the polytime functions; the witness programs are extracted from proofs as light affine lambda-calculus terms.

ICC and BSS. An extension of ICC to the BSS model was proposed by Bournez *et al.* in [BCdNM03a]: this article characterizes in particular by means of *safe recursion* the class FP(K) over an arbitrary structure. Recall that safe recursion was introduced by Bellantoni and Cook ([BC92]) as a restriction of primitive recursion based on the distinction between two classes of arguments (normal and safe) and characterizing the (classical) class FP. In the BSS case the recursion considered is on the structure of lists over K. Their approach was extended to other complexity classes such as PAR and the polynomial hierarchy in [BCdNM03b] thus demonstrating the relevance of ICC tools to the BSS framework.

Our goal and contribution. In the present work we use light affine logic and light affine lambda-calculus to provide a new characterization of the class $FP(K)$ of deterministic polynomial time functions over K. In the long term we wish to develop a theoretical language to write feasible algorithms on an arbitrary ring and to allow formal reasoning on these algorithms. We think that lambda-calculus and LAL offer two main advantages in this perspective:

- higher-order: in numerical analysis algorithms functions have a first-class status, and one naturally handles higher-order functionals; thus it is an important point to have a language which includes higher-order;

- proofs: computation over $\mathbb{R}$, $\mathbb{C}$ or other rings or fields is a framework in which we would certainly like to be able to manage together mathematical proofs and programs in an integrated way; Light linear logic is interesting in this respect because it provides a setting which can accommodate program extraction from proofs.

Finally, some semantic interpretations of Light linear logic have been given, both for semantics of formulas (phase spaces, [KOS03]) an for semantics of proofs (in games, [MO00] or coherent spaces [Bai04]). Thus the present work provides a first step from which semantic approaches for the study of BSS polytime functions can be considered.

Concretely, our extension of light lambda-calculus is very simple: to the type language (LAL) we just add a basic type for K and to the term language some constants for the elements of K and for the operations and relations of K (a bit as in the language PCF with the type of integers for instance). The contribution of the present paper is then to show the validity of this approach:

- we show that any term on lists over K in this language denotes an $FP(K)$ function;

- we show that BSS polytime machines can be simulated, hence all $FP(K)$ functions can be programmed.

Outline of the paper. In section 2 we recall Light affine logic and light affine lambda-calculus and in section 3 the BSS model. In section 4, we define our extension $\lambda_{LA\mathbb{K}}$ of light affine lambda-calculus to a structure K; then we show that the terms can be reduced in polynomial time (section 5) and conversely that all ptime BSS machines over K can be simulated in $\lambda_{LA\mathbb{K}}$ (section 6).

2 LAL and λ_{LAL}

The formulas of Intuitionistic Light affine logic (LAL) are given by the following grammar:

$$A, B := \alpha \mid A \multimap B \mid !A \mid \S A \mid \forall \alpha.A$$

The modalities !, §, called *exponentials* are used to control duplication. An erasure map $(.)^-$ from LAL formulas to system F types is given by:
$(A \multimap B)^- = A \to B$, $(!A)^- = (\S A)^- = A^-$, $(\forall \alpha.A)^- = \forall \alpha.A^-$.

Following Terui ([Ter01]), we consider λ_{LAL} a typed lambda-calculus with types of intuitionistic light affine linear logic. This calculus has explicit constructs for handling ! and §. Its terms are defined by the grammar:

$$t, u ::= x|\lambda x\, t|(t)u|!t|\mathtt{let}\, u\, \mathtt{be}\, !x\, \mathtt{in}\, t|\S t|\mathtt{let}\, u\, \mathtt{be}\, \S x\, \mathtt{in}\, t$$

In typing judgments, besides ordinary LAL formulas we will use !-*discharged* and §-*discharged* formulas, of the form $[A]_\dagger$ with respectively $\dagger =!$ or §. Discharged formulas have a temporary status, they cannot be applied any connective and are only a technical artifact to manage structural rules (contractions) in a convenient way.

The typing rules are now given on Figure 1.

Note that the typing rules are here given in a sequent calculus style (with right and left introduction rules); a natural deduction presentation could also have been used.

The most important rule to notice is $!_r$: as *Cntr* is performed only on !-discharged variables, during reduction only ! typed terms will be duplicated; the $!_r$ rule ensures that duplicable terms have *at most one occurrence* of free variable. This is one of the keys that ensure the polynomial bound for the reduction of these terms ([Ter01]). Another important point is a stratification property ensured by the ! and § connectives: in particular note that to ! discharge a variable x (thus making it contractible) one has to apply a $!_r$ or a $\S_r$ rules and add an exponential to the type of the term.

Actually, in [Ter01] terms are defined as a subclass of *pseudo-terms* satisfying some syntactical conditions. We could do the same here but as we will only consider typed terms this is not necessary (all well-typed pseudo-terms are terms).

The reduction relation is defined as the contextual, reflexive and transitive closure of the relation given on Figure 2. Actually the β rule is a linear beta reduction step and only the ! rule can cause duplications.

Terms of λ_{LAL} should in fact be seen as ordinary lambda-terms with extra information on sharing and stratification given by the ! and § constructs. By erasing this information from a term t we get an ordinary lambda-term t^- which denotes the same function as t:

$$\begin{array}{lcl}(!t)^- & = & (\S t)^- = t^-,\\ x^- & = & x, (\lambda x\, t)^- = \lambda x\, t^-, [(t)u]^- = (t)^- u^-,\end{array}$$

$$(\mathtt{let}\, u\, \mathtt{be}\, \dagger\, x\, \mathtt{in}\, t)^- = t^-[u^-/x].$$

If $\Gamma \vdash_{LAL} t : A$ we then have in system F: $(\Gamma)^- \vdash_F t : A^-$. Moreover if t is a λ_{LAL} term and $t \to t'$, then we have with ordinary beta reduction: $t^- \xrightarrow{\star} t'^-$

We could in fact instead of λ_{LAL} have used ordinary lambda-terms typed in DLAL (see [BT04]), a system which is essentially a fragment of LAL. The properties in the rest of this paper could have been proved in the same way.

2.1 Syntactic sugar: lambda calculus macro definitions

2.1.1 Tensor

We consider $\otimes$ as a defined construct. On types we set:

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

On terms we define:
$\mathtt{let}\, u\, \mathtt{be}\, x \otimes y\, \mathtt{in}\, t = (u)\lambda x \lambda y.t$
$t_1 \otimes t_2 = \lambda y (y) t_1\, t_2$.
Then we have the following typing rules:

$$\frac{x : A, y : B, \Gamma \vdash t : C}{z : A \otimes B, \Gamma \vdash \mathtt{let}\, z\, \mathtt{be}\, x \otimes y\, \mathtt{in}\, t : C}\ \otimes_l$$

$$\frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash t \otimes u : A \otimes B}\ \otimes_r.$$

and the reduction rule:

$$\mathtt{let}\, (u_1 \otimes u_2)\, \mathtt{be}\, (x_1 \otimes x_2)\, \mathtt{in}\, t \xrightarrow{(\otimes)} t[u_1/x_1, u_2/x_2].$$

$$\frac{}{x : A \vdash x : A}\ Id \qquad \frac{\Gamma_1 \vdash u : A \qquad x : A, \Gamma_2 \vdash t : C}{\Gamma_1, \Gamma_2 \vdash t[u/x] : C}\ Cut$$

$$\frac{\Gamma \vdash t : C}{\Delta, \Gamma \vdash t : C}\ Weak \qquad \frac{x : [A]_!, y : [A]_!, \Gamma \vdash t : C}{z : [A]_!, \Gamma \vdash t[z/x, z/y] : C}\ Cntr$$

$$\frac{\Gamma_1 \vdash u : A_1 \quad x : A_2, \Gamma_2 \vdash t : C}{\Gamma_1, y : A_1 \multimap A_2, \Gamma_2 \vdash t[(y)u/x] : C}\ \multimap_l \qquad \frac{x : A_1, \Gamma \vdash t : A_2}{\Gamma \vdash \lambda x\, t : A_1 \multimap A_2}\ \multimap_r$$

$$\frac{x : A[B/\alpha], \Gamma \vdash t : C}{x : \forall \alpha A, \Gamma \vdash t : C}\ \forall_l \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall \alpha A}\ \forall_r \quad (\alpha \text{ not free in } \Gamma)$$

$$\frac{x : [A]_!, \Gamma \vdash t : C}{y : !A, \Gamma \vdash \mathtt{let}\, y\, \mathtt{be}\, !x\, \mathtt{in}\, t : C}\ !_l \qquad \frac{x : B \vdash t : A}{x : [B]_! \vdash !t : !A}\ !_r \quad \text{with a possibly empty context.}$$

$$\frac{x : [A]_{\S}, \Gamma \vdash t : C}{y : \S A, \Gamma \vdash \mathtt{let}\, y\, \mathtt{be}\, \S x\, \mathtt{in}\, t : C}\ \S_l \qquad \frac{\Gamma, \Delta \vdash t : A}{[\Gamma]_!, [\Delta]_{\S} \vdash \S t : \S A}\ \S_r \quad \text{with } \Gamma \text{ and } \Delta \text{ possibly empty.}$$

Figure 1: LAL typing rules.

In the sequel we will use as a short-hand compound patterns such as for instance $x_1 \otimes x_2 \otimes x_3$ or $(\S x_1) \otimes x_2$, for which the `let` constructs are definable from the `let` constructs for !, §, $\otimes$.

Let us also denote by $\lambda x \otimes y.\, t$ the term $\lambda z\, \mathtt{let}\, z\, \mathtt{be}\, x \otimes y\, \mathtt{in}\, t$. Hence we have:

$$(\lambda x \otimes y.\, t) u \otimes v \to t[u/x, v/y].$$

2.1.2 Integers and Booleans encodings

Tally integers are given by the type: $N = \forall \alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$.

Booleans are defined by the type $Bool = \forall \alpha.(\alpha \multimap \alpha \multimap \alpha)$:

$\mathtt{true} = \lambda x \lambda y.y$ and $\mathtt{false} = \lambda x \lambda y.y$.

We define a term for conditional:

$\mathtt{if}\ b\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2$ $=$ $(((b)\lambda x_1 \ldots \lambda x_n.u_1)\lambda x_1 \ldots \lambda x_n.u_2) x_1 \ldots x_n$

with the typing rule:

$$\frac{\vdash b : Bool \quad \Gamma \vdash u_1 : A \quad \Gamma \vdash u_2 : A}{\Gamma \vdash \mathtt{if}\ b\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2 : A}\ ite$$

3 BSS Machines over $\mathbb{K}$

Recall that a (classical) Turing machine over a finite alphabet A and finite set of states Q is given by a function

$$\mu : Q \times \tilde{A} \to Q' \times \tilde{A} \times \{-1, 0, 1\}$$

where $\tilde{A} := A \cup \square$ and $Q' := Q \cup \{q_F^0, q_F^1\}$ with $q_F^0, q_F^1 \notin Q$ are respectively called rejecting and accepting state.

Let $(K, +, *, 0, 1)$ be a ring. A *structure* on K is a tuple:

$$\mathbb{K} = (K, op_1^{k_1}, \ldots, op_n^{k_n}, \rho_1^{s_1}, \ldots \rho_m^{s_m}),$$

where each $op_i^{k_i}$ is a polynomial function (operation) over K of arity k_i and each ρ_i is a predicate over K of arity s_i. We assume one of the predicates, say ρ_1, is the equality. In the case where K is a field the $op_i^{k_i}$ can be defined by rational functions instead of polynomials. Operators of arity 0 are constants.

$$(\lambda x.t)u \xrightarrow{(\beta)} t[u/x]$$
$$\texttt{let}\,!u\,\texttt{be}\,!x\,\texttt{in}\,t \xrightarrow{(!)} t[u/x]$$
$$\texttt{let}\,\S u\,\texttt{be}\,\S x\,\texttt{in}\,t \xrightarrow{(\S)} t[u/x]$$
$$\texttt{let}\,(\texttt{let}\,u_1\,\texttt{be}\,\dagger_1\,x\,\texttt{in}\,t_1)\,\texttt{be}\,\dagger_2\,y\,\texttt{in}\,t_2 \xrightarrow{(com1)} \texttt{let}\,u_1\,\texttt{be}\,\dagger_1\,x\,\texttt{in}\,(\texttt{let}\,t_1\,\texttt{be}\,\dagger_2\,y\,\texttt{in}\,t_2)$$
$$(\texttt{let}\,u_1\,\texttt{be}\,\dagger_1\,x\,\texttt{in}\,t_1)t_2 \xrightarrow{(com2)} \texttt{let}\,u_1\,\texttt{be}\,\dagger_1\,x\,\texttt{in}\,(t_1)t_2 \text{ where } \dagger_i =! \text{ or } \S, \text{ for } i = 1, 2$$

Figure 2: Reduction rules

Two examples of structures are:

$$\begin{aligned}\mathbb{K}_1 &= (\mathbb{R}, +, -, *, (c_i)_{i\in\mathbb{R}}, =, \leq),\\ \mathbb{K}_2 &= (\{0,1\}, \vee, \wedge, 0, 1, =).\end{aligned}$$

A *BSS machine* over $\mathbb{K}$ ([BCSS98]) is a generalization of Turing machines that we shall describe below.

For a given structure $\mathbb{K}$ we denote by $K_\infty = K^{\mathbb{Z}}$ and by

$$K^\infty = \bigcup_{i=1}^{\infty} K^m \qquad \text{where } K^m = \{(x_1, \ldots, x_m) | x_i \in K\}.$$

A machine has a finite set of states Q and for each state $q \in Q$ only one of the following kinds of actions can be performed:

- (computation) at this step we aim to compute the value of one of the operations op_i using the first k_i elements of K^∞, the result is then stored in place of the current position.
- (branch) at this step we aim to compute the value of one of the relations ρ_i using the first s_i elements of K^∞, the result is used to select a state.
- (shift) this last type of action a BSS machine can perform corresponds to the movement of the head of the machine (on to the left or on to the right).

Definition 1 *Given a structure $\mathbb{K}$, a machine over $\mathbb{K}$ is a function $\mu : Q \to \mathcal{F}$, where*

$$\mathcal{F} = \bigcup_{i\in\mathbb{N}} (\tilde{K}^i \to Q' \times \tilde{K} \times \{-1, 0, 1\}).$$

where $Q' = Q \cup \{q_F^1, q_F^0\}$ and $q_F^1, q_F^0 \notin Q$.

Every state q in Q' (also called node) can be of one of the following five types: computation, branch *or* shift *as described above,* input *or* output.

For every node in Q the corresponding action is determined by q, it depends on n_q elements of K and gives as a result a triple $Q' \times \tilde{K} \times \{-1, 0, 1\}$:

$$\mu(q) : \tilde{K}^{n_q} \to Q' \times \tilde{K} \times \{-1, 0, 1\}$$

- *(computation)*

  $$\mu(q)(k_1, \ldots, k_{n_q}) = (q', op_i(k_1, \ldots, k_{n_q}), 0)$$

  where op_i is determined by q;

- *(branch)*

  $$\mu(q)(k_1, \ldots, k_{n_q}) = (q_b, k_1, 0)$$

  where $b = \rho_i(k_1, \ldots, k_{n_q})$ and the relation ρ_i to be applied is determined by q;

- *(shift)*

  $$\mu(q)(k_1, \ldots, k_{n_q}) = (q', k_1, m_q)$$

  and q' only depends on q (note that here dependency on $k_1, \ldots, k_{n_q}$ is formal).

We have exactly one input node denoted by $q_0 \in Q$, and two distinguished output nodes denoted by $q_F^1 \in Q'$ and $q_F^0 \in Q'$.

All the actions modify the current configuration of the machine: let us define a configuration of the machine at a given time as a triple (f, p, q) where $f \in \mathbb{K}^{\mathbb{Z}}$, $p \in \mathbb{Z}$ and $q \in Q'$.

In the encoding part since we cannot represent by a term the infinitary f but only a finite part of it, we will represent such a configuration as a triple $\langle f^-, f^+, q\rangle$, where $f^- = (f(-1), f(-2), \ldots, f(-n^-))$ if $f(-k) = \Box$ for all $k \geq n^-$, and analogously $f^+ = (f(0), f(1), f(2), \ldots, f(n^+))$ if $f(k) = \Box$ for all $k \geq n^+$.

The initial configuration associated to $w \in K^\infty$ is $(f_w, 1, q_0)$ where $f_w \in K_\infty$ is

$$f_w(i) = \begin{cases} w_i & \text{for all } 1 \leq i \leq |w| \\ \Box & \text{otherwise.} \end{cases}$$

A transition from the configuration (f, p, q) gives the configuration $(f', p + m, q')$ if

$$\mu(q)(f(1), \ldots, f(n_q)) = (q', k, m)$$

and $f'(i) = f(i)$ for every $i \neq p$ and $f'(p) = k$.

Since μ is not defined on output nodes, they correspond to the end of the computation.

An input word w is *accepted* by μ if the machine starting from the input node evolves according to the transitions specified by its nodes and eventually reaches the q_F^1 output node.

A language $L \subset K^\infty$ is said to be *recognized* by μ if and only if it corresponds to the words which are accepted by μ.

A machine μ computes a function $g : K^\infty \to K^\infty$ if for any $w \in K^\infty$, μ evolves from the initial configuration $(f_w, 1, q_0)$ to a terminal configuration $(f_{g(w)}, 1, q_F^1)$ if f is defined on w and the computation does not end if f is not defined for w.

4 A light lambda-calculus for the structure $\mathbb{K}$

In order to manage arities we consider a variant of the definition of BSS machines; for a given structure $\mathbb{K}$, let us consider the maximal arity p for op_i and ρ_i in $\mathbb{K}$:

$$p = \max\big(\max_{1\leq i\leq n} k_i, \max_{1\leq i\leq m} s_i\big).$$

Then we consider an algebraic structure $\mathbb{K}^{up}$ with all the operations and relations of arity p defined as follows:

$$op_i'(x_1, \ldots, x_p) = op_i(x_1, \ldots, x_{k_i})$$

and

$$\rho_i'(x_1, \ldots, x_p) = \rho_i(x_1, \ldots, x_{s_i}).$$

For the algebra $\mathbb{K}^{up}$ we have a transition step with uniform number of elements of the tape: $\mu(q) : K^p \to Q' \times \tilde{K} \times \{-1, 0, 1\}$. Below we suppose $\mathbb{K}$ of uniform arity p.

We extend types of LAL with a basic type κ for elements of K. We add to the language constants $\star$ (used as empty list symbol), $\underline{k}$ for each $k \in K$ and $\mathtt{dup}$ (for duplication), $\mathtt{op}_i$ $(1 \leq i \leq n)$, ρ_i $(1 \leq i \leq m)$ for each operation and relation of the structure. The new term language $\lambda_{LA\,\mathbb{K}}$ is thus given by:

$$\begin{aligned} t, u ::= & x | \lambda x\, t | (t) u | \\ & !t | \mathtt{let}\ u\ \mathtt{be}\ !x\ \mathtt{in}\ t | \S t | \mathtt{let}\ u\ \mathtt{be}\ \S x\ \mathtt{in}\ t | \\ & t_1 \otimes t_2 | \mathtt{let}\ u\ \mathtt{be}\ x \otimes y\ \mathtt{in}\ t | \mathtt{dup} | \\ & \underline{k} | \star | \mathtt{op}_i | \mathtt{rho}_i, \end{aligned}$$

for every $k \in K$. The new typing rules for the constants are given on Figure 3 where κ^p denotes $\underbrace{\kappa \otimes \cdots \otimes \kappa}_{p}$.

For these constants we consider associated reduction rules given on Figure 4. We will denote by $\to$ the resulting new reduction relation and by $\xrightarrow{\star}$ its reflexive and transitive closure. Note the reduction (*dup*) is performed only when the argument is a value. Given a type A we will consider the usual type for lists of elements of A: $\mathcal{L}(A) = \forall\alpha.!(A \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. So for lists over K we have the type $\mathcal{L}(\kappa)$. We denote by *nil* the empty list. Recall that $\mathcal{L}(A)$ allows defining for any B a $\mathtt{fold}_B$ map with type:

$$\mathtt{fold}_B : !(A \multimap B \multimap B) \multimap \S B \multimap \mathcal{L}(A) \multimap \S B.$$

As in previous work on light logics (see [Gir98] and [AR02]) we will represent functions on lists by terms of type $\mathcal{L}(A) \multimap \S^n \mathcal{L}(A)$, where n is an integer.

5 The calculus $\lambda_{LA\,\mathbb{K}}$ is polytime

We want to show that terms of $\lambda_{LA\,\mathbb{K}}$ can be reduced in a polynomial number of steps. For that we adapt

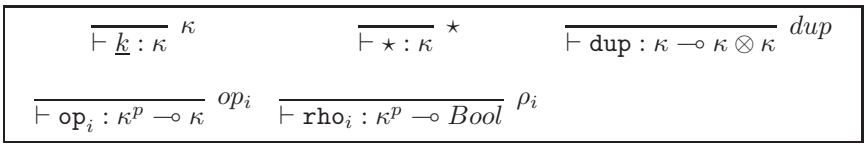

Figure 3: Typing rules for constants.

$(\mathtt{dup})\underline{k} \xrightarrow{(dup)} \underline{k} \otimes \underline{k}$

$(\mathtt{op}_i)\underline{k_1} \dots \underline{k_p} \xrightarrow{(op)} \underline{k}$ if $op_i(k_1, \dots, k_{k_i}) = k$,

$(\mathtt{rho}_i)\underline{k_1} \dots \underline{k_p} \xrightarrow{(rho)} b$ where if $\rho_i(k_1, \dots, k_{s_i})$ holds (resp. does not hold) then $b = \mathtt{true}$ (resp. $b = \mathtt{false}$)

Figure 4: Reduction rules for constants.

Terui's proof of weak polystep normalization for light affine lambda-calculus in [Ter01], which in turn follows [Gir98].

We consider a measure for terms of $\lambda_{LA\,\mathbb{K}}$ defined by:

$$
\begin{array}{ll}
|x| = 1, & \\
|\underline{k}| = 1, & |\lambda x.t| = |t| + 1, \\
|op_i| = 2, & |(t)u| = |t| + |u| + 1, \\
|\mathtt{rho}_i| = 4, & |\,\dagger\, t| = |t| + 1, \text{ for } \dagger = !, \S \\
|\mathtt{dup}| = 5, & |\mathtt{let}\, u\, \mathtt{be}\, \dagger\, x\, \mathtt{in}\, t| = |t| + |u| + 1, \\
 & \qquad \text{for } \dagger = !, \S
\end{array}
$$

We denote by $size(t)$ the number of nodes of the syntactic tree of t; therefore $size(t) \leq |t|$. We have:

Lemma 1 *If $t \xrightarrow{(r)} t'$ and $(r) \neq (!), (com\, i)$ then $|t'| < |t|$. If $(r) = (com\, i)$ then $|t'| = |t|$.*

Let $t \xrightarrow{\sigma\, \star} t'$ be a reduction sequence of t'. The *length* $|\sigma|$ of the reduction σ is its number of steps.

We say a reduction of a term t is *standard* if it is obtained in the following way:

$$t = t_0 \xrightarrow{\star} t_1 \xrightarrow{\star} \dots t_i \xrightarrow{\star} t_{i+1} \dots \xrightarrow{\star} t_n$$

where:

- sequences $t_{2j} \xrightarrow{\star} t_{2j+1}$ consist only of non (!) reduction steps at depth j,
- sequences $t_{2j+1} \xrightarrow{\star} t_{2j+2}$ consist only of (!) reduction steps at depth j.

With the notion of measure chosen the proofs of the other lemmas of [Ter01] remain valid for $\lambda_{LA\,\mathbb{K}}$ and we get in the same way:

Theorem 1 *Let t be a $\lambda_{LA\,\mathbb{K}}$ term of depth d and σ be a standard reduction $t \xrightarrow{\sigma\, \star} u$; then $|u| \leq |t|^2$ and $|\sigma| \leq |t|^{2^{d+1}}$.*

Moreover, each reduction step on a term v can be simulated on a BSS machine over $\mathbb{K}$ in a number of steps proportional to $size(v)^2$, so to $|v|^2$. Hence each step of the standard reduction σ can be simulated in time $O((|t|^{2^d})^2)$, so $O(|t|^{2^{d+1}})$. Therefore σ can be simulated in time $O(|t|^{2^{d+1}}.|t|^{2^{d+1}}) = O(|t|^{2^{d+2}})$, so polynomial in $|t|$ (at fixed depth d).

It follows then that:

Theorem 2 *If f is a function on lists over $\mathbb{K}$ representable by a $\lambda_{LA\,\mathbb{K}}$ term, then f belongs to the class $FP(\mathbb{K})$.*

6 Encoding of ptime BSS machines

The first difference with Roversi's encoding of classical Turing machines ([AR02]) is due to the fact that

in the BSS-model the choice between branching, computation or shift depends exclusively upon the current state; this simplifies the construction of the transition function which usually is given as a bidimensional table, whereas here we encode it as an array whose elements are selected by the current state.

6.1 States

If Q is the set of states of the machine μ:

$$Q = \{q_0, \ldots, q_d\}$$

we encode the state q_i by the term $\mathtt{q}_i = \lambda x_0 \lambda x_1 \ldots \lambda x_d \lambda v(x_i)v$. In fact, the term $\mathtt{q}_i$ is a selector to extract from a table a function of type

$$\alpha \otimes \kappa^p \otimes \kappa^p \otimes \alpha \multimap \alpha \otimes \alpha \otimes Q.$$

This function will be obtained by composing either the terms representing op_i or ρ_i with a term which transforms the current state in the next one.

So we have that the type of states is

$$Q = \forall\alpha\forall\beta(\underbrace{(\alpha \multimap \beta) \otimes \cdots \otimes (\alpha \multimap \beta)}_{d-times} \otimes \alpha) \multimap \beta.$$

6.2 Configurations and transitions

As already introduced in section 3, a configuration is a triple given by the left (half)tape (or negative tape) by the right tape (or positive tape) and by the current state.

$$\langle f^-, f^+, q\rangle = \lambda g\lambda x\lambda x'((g)k_1^-)((g)k_2^-)\ldots((g)k_{n^-}^-)x \otimes \\ \otimes (((g)k_1^+)((g)k_2^+)\ldots((g)k_{n^+}^+)x' \otimes q_i$$

when $f^- = (k_1^-, k_2^-, \ldots, k_{n^-}^-)$, $f^+ = (k_1^+, k_2^+, \ldots, k_{n^+}^+)$ with $k_i^\epsilon : \kappa$ and $q : Q$. The type of configurations is then:

$$C = \forall\alpha!(\kappa \multimap \alpha \multimap \alpha) \multimap \S\alpha \multimap \S\alpha \multimap \S(\alpha \otimes \alpha \otimes Q)$$

Note that to execute an action the machine might have to look ahead on the tape the next p squares: in the case of (computation) or (branch) action this is needed to fetch the p arguments on which to apply op_i or ρ_i. It is convenient for that to have a *window* of length $2p$ of elements directly accessible (representing the p elements on the right and on the left of the head). Such a window will have type $\kappa^p \otimes \kappa^p$.

Thus for the encoding we proceed in two steps:

- first, from a configuration we produce a *configuration with window*, of type:
  $CW = \forall\alpha!(\kappa \multimap \alpha \multimap \alpha) \multimap \S\alpha \multimap \S\alpha \multimap \S(\alpha \otimes \kappa^p \otimes \kappa^p \otimes \alpha \otimes Q)$.
- second, on a configuration with window we perform a transition step (with an action determined by the state) which produces a new configuration.

The first step will be done by a term $\mathtt{c2cw} : C \multimap CW$; using this term in the second step we will give a term $\mathtt{c2c} : C \multimap C$.

Let us first consider the term $c2cw : C \multimap CW$: applied to a configuration $\langle f^-, f^+, q\rangle$, `c2cw` will yield a configuration with window:

$$\langle h^-, (k_1^-, k_2^-, \ldots, k_{p^-}^-), (k_1^+, k_2^+, \ldots, k_{p^+}), h^+, q\rangle$$

where:

$$f^- = (k_1^-, k_2^-, \ldots, k_{n^-}^-), \quad f^+ = (k_1^+, k_2^+, \ldots, k_{n^+}^+),$$
$$h^- = (k_2^-, \ldots, k_{n^-}^-), \quad h^+ = (k_2^+, \ldots, k_{n^+}^+),$$

if $n^+ \geq p$ and $n^- \geq p$. In the case where $n^+ < p$ or $n^- < p$ the missing values for the window are replaced by $\star$.

The term `c2cw` is defined by an iteration:

$$\mathtt{c2cw} = \lambda c\lambda g\lambda x\lambda x'.\mathtt{let}(c)\mathtt{step}\,\mathtt{base}^-\,\mathtt{base}^+ \\ \mathtt{be}\,\S((b_1 \otimes \vec{k_1} \otimes l_1) \otimes (b_2 \otimes \vec{k_2} \otimes l_2)) \otimes q \\ \mathtt{in}\,\S(l_1 \otimes \vec{k_1} \otimes \vec{k_2} \otimes l_2 \otimes q) \quad (1)$$

where $\vec{k_i}$ denotes a tensor of p elements of type κ:

$$\vec{k_i} = k_{i1} \otimes \cdots \otimes k_{ip},$$

and

$$\mathtt{step} = \lambda k''\lambda b \otimes \vec{k_1} \otimes l.\mathtt{let}\,(\mathtt{dup})k_{11}\,\mathtt{be}\,c \otimes d\,\mathtt{in} \\ \mathtt{true} \otimes k'' \otimes c \otimes k_{12} \cdots \otimes k_{1(p-1)} \otimes \\ (\mathtt{if}\ b\ \mathtt{then}\ (g)d\ \mathtt{else}\ I)l$$

is of type $\texttt{step} : !(\kappa \multimap (Bool \otimes \kappa^p \otimes \beta \multimap Bool \otimes \kappa^p \otimes \beta))$. The terms $\texttt{base}^+ = \S(\texttt{false} \otimes \star^p \otimes x')$ and $\texttt{base}^- = \S(\texttt{false} \otimes \star^p \otimes x)$ are of type $\texttt{base}^+ : \S(Bool \otimes \kappa^p \otimes \beta)$.

Note that duplication of values of type κ with `dup` has been used in the term `c2cw` (via `step`) to build the window. This only comes from the fact that a value written on the tape used for an operation or a test remains written and can be used another time. This is actually the only place in the encoding where `dup` is used.

Now, once given a configuration with window, to perform a transition step we need a term which will, depending on the state, select the right action to perform:

$$\texttt{next\_conf} : Q \multimap \alpha \otimes \kappa^p \otimes \kappa^p \otimes \alpha \multimap \alpha \otimes \alpha \otimes Q$$

This term simply uses the definition of states:

$$\texttt{next\_conf} = \lambda q.(q)t_1 \dots t_d,$$

where t_j is a term corresponding to the action $\mu(q_j)$ of the transition table. To define the t_js we have to consider the three possible transitions in the BSS-machine:

1. (computation) the top of the positive part of the tape is replaced with the application of an operation op_i to the first p elements of the tape, q' is the new state of the machine:

$$\begin{aligned} t_j = &\lambda l_1 \otimes \vec{k_1} \otimes \vec{k_2} \otimes l_2. \\ &((g)k_{11})l_1 \otimes \\ &\otimes ((g)(\texttt{op}_i)k_{21} \dots k_{2p})l_2 \otimes q' \end{aligned}$$

2. (branch) the branch case, the machine chooses the next state q_1 or q_2 depending on the result of the evaluation of the relation ρ_i with the first p elements of the positive tape as arguments

$$\begin{aligned} t_j = &\lambda l_1 \otimes \vec{k_1} \otimes \vec{k_2} \otimes l_2. \\ &((g)k_{11})l_1 \otimes \\ &\otimes ((g)k_{21})l_2 \otimes \\ &\otimes \texttt{if } (\texttt{rho}_i)k_{21} \dots k_{2p} \texttt{ then } q_1 \texttt{ else } q_2 \end{aligned}$$

3. (shift) the left shift consists in moving the first element of the negative tape to the top of the positive one with q' as the new state:

$$\begin{aligned} t_j = &\lambda l_1 \otimes \vec{k_1} \otimes \vec{k_2} \otimes l_2. \\ &l_1 \otimes \\ &\otimes ((g)k_{11})((g)k_{21})l_2 \otimes q' \end{aligned}$$

analogously we do for the right shift:

$$\begin{aligned} t_j = &\lambda l_1 \otimes \vec{k_1} \otimes \vec{k_2} \otimes l_2. \\ &((g)k_{21})((g)k_{11})l_1 \otimes \\ &\otimes l_2 \otimes q' \end{aligned}$$

Finally, using the terms `c2cw` and `next_conf` we define the term `c2c` which performs a transition step on a configuration:

$$\begin{aligned} \texttt{c2c} = \lambda c \lambda g \lambda x \lambda x' &\texttt{let}(\texttt{c2cw})cgxx' \\ &\texttt{be } \S(l_1 \otimes \vec{k_1} \otimes \vec{k_2} \otimes l_2 \otimes q) \\ &\texttt{in } \S((\texttt{next\_conf})q)(l_1 \otimes \vec{k_1} \otimes \vec{k_2} \otimes l_2) \end{aligned} \quad (2)$$

Note that g, x, x' are bound by the `c2c` abstractions.

6.3 Completing the encoding.

Just as in [AR02] one can define the following terms:

$$\begin{aligned} &\texttt{length}: && \mathcal{L}(\kappa) \multimap \S N, \\ &\texttt{init}: && \mathcal{L}(\kappa) \multimap C, \\ &\texttt{extract}: && C \multimap \mathcal{L}(\kappa). \end{aligned}$$

The term `length` computes the length of the list as a tally integer; `init` maps a list l onto the corresponding initial configuration $\langle nil, l, q_0 \rangle$; `extract` recovers from a configuration $\langle f^-, f^+, q \rangle$ the list corresponding to f^+.

Now, given an input l of type $\mathcal{L}(\kappa)$ for the machine, we will need to use l for two purposes:

(i) to produce the initial configuration (with `init`),

(ii) to yield an integer (its length) n, from which the time bound for the machine will be computed.

For that it is easy to define using `fold` a term $\texttt{Ilength} : \mathcal{L}(\kappa) \multimap \S(\mathcal{L}(\kappa) \otimes N)$ such that:

$$(\texttt{Ilength})l \xrightarrow{\star} \S(l \otimes n),$$

where n is the length of l.

Recall that:

Lemma 2 ([AR02]) *For any polynomial P in $\mathbb{N}[X]$ there exists an integer k and a term $t_P : N \multimap \S^k N$ such that t_P represents P.*

Now, let μ be a polytime BSS machine with polynomial P. One can define a term u simulating μ in the following way:

- apply `Ilength`, and then `init` to the l.h.s. result to get a configuration c_0, and t_P to the r.h.s. result to get an integer $m = P(n)$ (where n is the length of the input);
- use m to iterate the term `c2c`, m times starting from c_0 and get a configuration c_1;
- apply `extract` to c_1.

Typing in a suitable way this procedure one obtains a term $u : \mathcal{L}(\kappa) \multimap \S^d\mathcal{L}(\kappa)$. We thus have:

Theorem 3 *For any function f in $FP(\mathbb{K})$, there exists an integer d and a term u of $\lambda_{LA\mathbb{K}}$ with type $\mathcal{L}(\kappa) \multimap \S^d\mathcal{L}(\kappa)$ representing f.*

7 Conclusions

We have presented an extension of light affine lambda-calculus to computation on an arbitrary ring structure $\mathbb{K}$. The definition of this extension is quite natural and it characterizes the BSS class $FP(\mathbb{K})$ in the same way light affine lambda-calculus characterized the classical class FP. Compared to the characterization by safe recursion from [BCdNM03a] our approach offers the advantage of integrating higher-order constructs which are likely to be useful in describing numerical analysis algorithms. We plan to examine some programming examples of algorithms in our calculus. It would also be interesting to see if other calculi which characterize FP in the classical setting and have higher-order such as those of [Laf04, Hof00, BNS00] can be extended in the same way to the BSS setting.

References

[AR02] A. Asperti and L. Roversi. Intuitionistic Light Affine Logic. *ACM Transactions on Computational Logic*, 3(1):137–175, 2002.

[Asp98] A. Asperti. Light affine logic. In *Proceedings LICS'98*, pages 300–308. IEEE Computer Society, 1998.

[Bai04] P. Baillot. Stratified coherence spaces: a denotational semantics for light linear logic. *Theoretical Computer Science*, 318(1-2):29–55, 2004.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[BCdNM03a] Olivier Bournez, Felipe Cucker, Paulin Jacobé de Naurois, and Jean-Yves Marion. Safe recursion over an arbitrary structure. sequential and parallel polynomial time. In *Proceedings of FoSSaCS 2003*, number 2620 in LNCS, pages 185–199. Springer, 2003.

[BCdNM03b] Olivier Bournez, Felipe Cucker, Paulin Jacobé de Naurois, and Jean-Yves Marion. Safe recursion over an arbitrary structure:PAR, PH and DPH. *ENTCS*, 90:3–14, 2003. Proceedings of ICC'02.

[BCSS98] L. Blum, F. Cucker, M. Shub, and S. Smale. *Compexity and Real Computation*. Springer-Verlag, 1998.

[BNS00] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3):17–30, 2000.

[BSS89] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, (21):1–46, 1989.

[BT04] P. Baillot and K. Terui. Light types for polynomial time computation in lambda-calculus. In *Proceedings of LICS'04*, 2004.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir98] J.-Y. Girard. Light linear logic. *Information and Computation*, 143:175–204, 1998.

[Hof99] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *"Proceedings of the 14^{th} Symposium on Logic in Computer Science"*, pages 464–473. IEEE Computer Society, 1999.

[Hof00] M. Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3):113–166, 2000.

[Jon01] N. Jones. The expressive power of higher order types. *Journal of Functional Programing*, 11:55–94, 2001.

[KOS03] M.I. Kanovich, M. Okada, and A. Scedrov. Phase semantics for light linear logic. *Theoretical Computer Science*, 294(3):525–549, 2003.

[Laf04] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1–2):163–180, 2004.

[Lei94] D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.

[LM93] D. Leivant and J.-Y. Marion. Lambda-calculus characterisations of polytime. *Fundamenta Informaticae*, 19:167–184, 1993.

[MM00] J.-Y. Marion and J.-Y. Moyen. Efficient first order functional interpreter with time bound certification. In *Proceedings LPAR*, volume 1955 of *LNCS*. Springer, 2000.

[MO00] A. S. Murawski and C.-H.L. Ong. Discreet games, light affine logic and ptime computation. In *Proceedings of CSL'00*, LNCS. Springer, 2000.

[Poi95] B. Poizat. *Les Petits Cailloux*. Aléas, 1995.

[Rov00] L. Roversi. Light affine logic as a programming language: a first contribution. *International Journal of Foundations of Computer Science*, 11(1), 2000.

[Ter01] K. Terui. Light Affine Lambda-calculus and polytime strong normalization. In *Proceedings LICS'01*. IEEE Computer Society, 2001.

[Ter04] K. Terui. Light affine set theory: a naive set theory of polynomial time. *Studia Logica*, 77:9–40, 2004.