An Elementary Affine λ-Calculus with Multithreading and Side Effects

Antoine Madet, Roberto M. Amadio

▶ To cite this version:

Antoine Madet, Roberto M. Amadio. An Elementary Affine λ-Calculus with Multithreading and Side Effects. C.-H. Luke Ong. Typed Lambda Calculi and Applications 10th International Conference, Jun 2011, Novi Sad, Serbia. Springer, 6690, pp.138-152, 2011, Lecture Notes in Computer Science. <10.1007/978-3-642-21691-6_13>. <hal-00610012>

HAL Id: hal-00610012

https://hal.archives-ouvertes.fr/hal-00610012

Submitted on 26 Sep 2012

An Elementary Affine λ-calculus with Multithreading and Side Effects[⋆]

Antoine Madet and Roberto M. Amadio

Laboratoire PPS, Université Paris Diderot
{madet,amadio}@pps.jussieu.fr

Abstract. Linear logic provides a framework to control the complexity of higher-order functional programs. We present an extension of this framework to programs with multithreading and side effects focusing on the case of elementary time. Our main contributions are as follows. First, we introduce a modal call-by-value λ-calculus with multithreading and side effects. Second, we provide a combinatorial proof of termination in elementary time for the language. Third, we introduce an elementary affine type system that guarantees the standard subject reduction and progress properties. Finally, we illustrate the programming of iterative functions with side effects in the presented formalism.

Key words: Elementary Linear logic. Resource Bounds. Lambda Calculus. Regions. Side Effects.

1 Introduction

There is a well explored framework based on Linear Logic to control the complexity of higher-order functional programs. In particular, *light logics* [11,10,3] have led to a polynomial light affine λ-calculus [14] and to various type systems for the standard λ-calculus guaranteeing that a well-typed term has a bounded complexity [9,8,5]. Recently, this framework has been extended to a higher-order process calculus [12] and a functional language with recursive definitions [4]. In another direction, the notion of *stratified region* [7,1] has been used to prove the termination of higher-order multithreaded programs with side effects.

Our general goal is to extend the framework of light logics to a higher-order functional language with multithreading and side effects by focusing on the case of elementary time [10]. The key point is that termination does not rely anymore on stratification but on the notion of depth which is standard in light logics. Indeed, light logics suggest that complexity can be tamed through a fine analysis of the way the depth of the occurrences of a λ-term can vary during reduction.

Our core functional calculus is a λ-calculus extended with a constructor '!' (the modal operator of linear logic) marking duplicable terms and a related $\mathsf{let}\,!$

[⋆] Work partially supported by project ANR-08-BLANC-0211-01 "COMPLICE" and the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881 (project CerCo).

destructor. The depth of an occurrence in a λ-term is the number of $!'s$ that must be crossed to reach the occurrence. In Section 2, following previous work on an affine-intuitionistic system [2], we extend this functional core with parallel composition and operations producing side effects on an 'abstract' notion of state. In Section 3, we analyse the impact of side-effects operations on the depth of the occurrences. Based on this analysis, we propose a formal system called *depth system* that controls the depth of the occurrences and which is a variant of a system proposed in [14]. In Section 4, we show that programs well-formed in the depth system are guaranteed to terminate in elementary time. The proof is based on an original combinatorial analysis of the depth system. In particular, as a corollary of this analysis one can derive an elementary bound for the functional fragment under an arbitrary reduction strategy ([10] assumes a specific reduction strategy while [14] relies on a standardization theorem). In Section 5, we refine the depth system with a second order (polymorphic) elementary affine type system and show that the resulting system enjoys subject reduction and progress (besides termination in elementary time). Finally, in Section 6, we discuss the expressivity of the resulting type system. On the one hand we check that the usual encoding of elementary functions goes through. On the other hand, and more interestingly, we provide examples of iterative (multithreaded) programs with side effects. The λ-calculi introduced are summarized in Table 1.1. For each concurrent language there is a corresponding functional fragment and each language (functional or concurrent) refines the one on its left hand side. The elementary complexity bounds are obtained for the $\lambda^{!}_{\delta}$ and $\lambda^{!\mathsf{R}}_{\delta}$ calculi while the progress property and the expressivity results refer to their typed refinements $\lambda^{!}_{EA}$ and $\lambda^{!\mathsf{R}}_{EA}$, respectively. Proofs are available in the technical report [13].

Functional	$\lambda^{!} \supset \lambda^{!}_{\delta} \supset \lambda^{!}_{EA}$
$\cap$	
Concurrent	$\lambda^{!\mathsf{R}} \supset \lambda^{!\mathsf{R}}_{\delta} \supset \lambda^{!\mathsf{R}}_{EA}$

Table 1.1. Overview of the λ-calculi considered

2 A Modal λ-calculus with Multithreading and Regions

In this section we introduce a call-by-value modal λ-calculus endowed with parallel composition and operations to read and write *regions*. We call it $\lambda^{!\mathsf{R}}$. A region is an *abstraction* of a set of dynamically generated values such as imperative references or communication channels. We regard $\lambda^{!\mathsf{R}}$ as an abstract, highly non-deterministic language which entails complexity bounds for more concrete languages featuring references or channels (we will give an example of such a language in Section 6). To this end, it is enough to map the dynamically generated values to their respective regions and observe that the reductions in the concrete languages are simulated in $\lambda^{!\mathsf{R}}$ (see, *e.g.*, [2]). The purely functional fragment, called $\lambda^{!}$, is very close to the *light affine* λ*-calculus* of Terui [14] where the paragraph modality '§' used for polynomial time is dropped and where the '!' modality is relaxed as in elementary linear logic [10].

2.1 Syntax

The syntax of the language is described in Table 2.1. We have the usual set

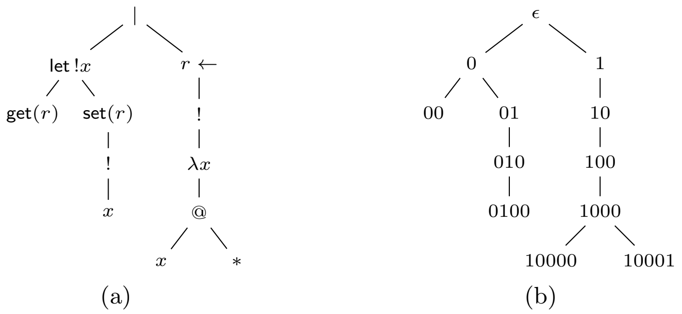
$x, y, \ldots$	(Variables)
$r, r', \ldots$	(Regions)
$V ::= * \mid r \mid x \mid \lambda x.M \mid !V$	(Values)
$M ::= V \mid MM \mid !M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ M$	
$\mathsf{set}(r, V) \mid \mathsf{get}(r) \mid (M \mid M)$	(Terms)
$S ::= (r \leftarrow V) \mid (S \mid S)$	(Stores)
$P ::= M \mid S \mid (P \mid P)$	(Programs)
$E ::= [\,] \mid EM \mid VE \mid !E \mid \mathsf{let}\ !x = E\ \mathsf{in}\ M$	(Evaluation Contexts)
$C ::= [\,] \mid (C \mid P) \mid (P \mid C)$	(Static Contexts)

Table 2.1. Syntax of programs: $\lambda^{!\mathsf{R}}$

of variable $x, y, \ldots$ and a set of regions $r, r', \ldots$. The set of values V contains the unit constant $*$, variables, regions, λ-abstraction and modal values $!V$ which are marked with the *bang* operator '!'. The set of terms M contains values, application, modal terms $!M$, a $\mathsf{let}\,!$ operator, $\mathsf{set}(r, V)$ to write the value V at region r, $\mathsf{get}(r)$ to fetch a value from region r and $(M \mid N)$ to evaluate M and N in parallel. A store S is the composition of several stores $(r \leftarrow V)$ in parallel. A program P is a combination of terms and stores. Evaluation contexts follow a call-by-value discipline. Static contexts C are composed of parallel compositions. Note that stores can only appear in a static context, thus $M(M' \mid (r \leftarrow V))$ is not a legal term.

We define $!^0 M = M$, $!^{n+1} M = !(!^n M)$, $!^n(P \mid P) = (!^n P \mid !^n P)$, and $!^n(r \leftarrow V) = (r \leftarrow V)$. In the terms $\lambda x.M$ and $\mathsf{let}\ !x = N\ \mathsf{in}\ M$ the occurrences of x in M are bound. The set of free variables of M is denoted by $\mathsf{FV}(M)$. The number of free occurrences of x in M is denoted by $\mathsf{FO}(x, M)$. $M[V/x]$ denotes the term M in which each free occurrence of x has been substituted by the value V (we insist for substituting values because in general the language is not closed under arbitrary substitutions). As usual, we abbreviate $(\lambda z.N)M$ with $M; N$, where z is not free in N.

Each program has an *abstract syntax tree* as exemplified in Figure 1(a). A path starting from the root to a node of the tree denotes an *occurrence* of the program that is denoted by a word $w \in \{0,1\}^*$ (see Figure 1(b)).



(a) (b)

Fig. 2.1. Syntax tree and addresses of $P = \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !x) \mid (r \leftarrow !(\lambda x.x*))$

$$\begin{array}{rcll}
P \mid P' & \equiv & P' \mid P & \text{(Commutativity)} \\
(P \mid P') \mid P'' & \equiv & P \mid (P' \mid P'') & \text{(Associativity)}
\end{array}$$

$$\begin{array}{lll}
E[(\lambda x.M)V] & & \rightarrow E[M[V/x]] \\
E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M] & & \rightarrow E[M[V/x]] \\
E[\mathsf{set}(r, V)] & & \rightarrow E[*] \qquad\quad \mid (r \leftarrow V) \\
E[\mathsf{get}(r)] & \mid (r \leftarrow V) & \rightarrow E[V] \\
E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] & \mid (r \leftarrow !V) & \rightarrow E[M[V/x]] \mid (r \leftarrow !V)
\end{array}$$

Table 2.2. Semantics of $\lambda^{!\mathsf{R}}$ programs

2.2 Operational Semantics

The operational semantics of the language is described in Table 2.2. Programs are considered up to a structural equivalence $\equiv$ which is the least equivalence relation preserved by static contexts, and which contains the equations for α-renaming and for the commutativity and associativity of parallel composition. The reduction rules apply modulo structural equivalence and in a static context C. In the sequel, $\overset{*}{\rightarrow}$ denotes the reflexive and transitive closure of $\rightarrow$.

The let ! operator is 'filtering' modal terms and 'destructs' the bang of the value $!V$ after substitution. When writing to a region, values are accumulated rather than overwritten (remember that $\lambda^{!\mathsf{R}}$ is an abstract language that can simulate more concrete ones where values relating to the same region are associated with distinct addresses). On the other hand, reading a region amounts to select non-deterministically one of the values associated with the region. We distinguish two rules to read a region. The first *consumes* the value from the store, like when reading a communication channel. The second *copies* the value from the store, like when reading a reference. Note that in this case the value read must be duplicable (of the shape $!V$).

Example 1. Program P of Figure 2.1 reduces as follows:

$$P \quad \rightarrow \quad \mathsf{set}(r, !(\lambda x.x*)) \mid (r \leftarrow !(\lambda x.x*)) \quad \rightarrow \quad * \mid (r \leftarrow !(\lambda x.x*)) \mid (r \leftarrow !(\lambda x.x*))$$

3 Depth System

In this section, we analyse the interaction between the depth of the occurrences and side effects. This leads to the definition of a depth system and the notion of *well-formed* program. As a first step, we introduce a naive definition of *depth*.

Definition 1 (naive depth). *The* depth $d(w)$ *of an occurrence* w *is the number of* ! *labels that the path leading to the end node crosses. The depth* $d(P)$ *of a program* P *is the maximum depth of its occurrences.*

With reference to Figure 2.1, $d(0100) = d(100) = d(1000) = d(10000) = d(10001) = 1$, whereas other occurrences are at depth 0. In particular, occurrences 010 and 10 are at depth 0; what matters in computing the depth of an occurrence is the number of !'s that precede strictly the end node. Thus $d(P) = 1$.

By considering that deeper occurrences have less weight than shallow ones, the usual proof of termination in elementary time [10] relies on the observation that when reducing a redex at depth i the following holds:

(1) the depth of the term does not increase,
(2) the number of occurrences at depth $j < i$ does not increase,
(3) the number of occurrences at depth i strictly decreases,
(4) the number of occurrences at depth $j > i$ may be increased by a multiplicative factor k bounded by the number of occurrences at depth $i + 1$.

If we consider the functional core of our language (*i.e.* by removing all operators dealing with regions, stores and multithreading), it is not difficult to check that the properties above can be guaranteed by the following requirements: (1) in $\lambda x.M$, x may occur at most once in M and at depth 0, (2) in $\mathsf{let}\ !x = M\ \mathsf{in}\ N$, x may occur arbitrarily many times in N and at depth 1.

However, we observe that side effects may increase the depth or generate occurrences at lower depth than the current redex, which violates Property (1) and (2) respectively. Then to find a suitable notion of depth, it is instructive to consider the following program examples where $M_r = \mathsf{let}\ !z = \mathsf{get}(r)\ \mathsf{in}\ !(z*)$.

$$\begin{array}{llll}(A) & E[\mathsf{set}(r, !V)] & (B) & \lambda x.\mathsf{set}(r, x); !\mathsf{get}(r) \\ (C) & !(M_r) \mid (r \leftarrow !(\lambda y.M_{r'})) \mid (r' \leftarrow !(\lambda y.*)) & (D) & !(M_r) \mid (r \leftarrow !(\lambda y.M_r))\end{array}$$

(A) Suppose the occurrence $\mathsf{set}(r, !V)$ is at depth $\delta > 0$ in E. Then when evaluating such a term we always end up in a program of the shape $E[*] \mid (r \leftarrow !V)$ where the occurrence $!V$, previously at depth δ, now appears at depth 0. This contradicts Property (2).

(B) If we apply this program to V we obtain $!V$, hence Property (1) is violated because from a program of depth 0, we reduce to a program of depth 1. We remark that this is because the read and write operations do not execute at the same depth.

(C) According to our definition, this program has depth 2, however when we reduce it we obtain a term $!^3*$ which has depth 3, hence Property (1) is violated. This is because the occurrence $\lambda y.M_{r'}$ originally at depth 1 in the store, ends up at depth 2 in the place of z applied to $*$.

(D) If we accept circular stores, we can even write diverging programs whose depth is increased by 1 every two reduction steps.

Given these remarks, the rest of this section is devoted to a revised notion of depth and to a related set of inference rules called depth system. Every program which is valid in the depth system will terminate in elementary time. First, we introduce the following contexts:

$$\Gamma = x_1 : \delta_1, \ldots, x_n : \delta_n \qquad\qquad R = r_1 : \delta_1, \ldots, r_n : \delta_n$$

where δ_i is a natural number. We write $dom(\Gamma)$ and $dom(R)$ for the sets $\{x_1, \ldots, x_n\}$ and $\{r_1, \ldots, r_n\}$ respectively. We write $R(r_i)$ for the depth δ_i associated with r_i in the context R. Then, we revisit the notion of depth as follows.

Definition 2 (revised depth). *Let P be a program, R a region context where $dom(R)$ contains all the regions of P and $d_n(w)$ the naive depth of an occurrence w of P. If w does not appear under an occurrence $r \leftarrow$ (a store), then the revised depth $d_r(w)$ of w is $d_n(w)$. Otherwise, $d_r(w)$ is $R(r) + d_n(w)$. The revised depth $d_r(P)$ of the program is the maximum revised depth of its occurrences.*

Note that the revised depth is relative to a fixed region context. In the sequel we write $d(\_)$ for $d_r(\_)$. On functional terms, this notion of depth is equivalent to the one given in Definition 1. However, if we consider the program of Figure 2.1, we now have $d(10) = R(r)$ and $d(100) = d(1000) = d(10000) = d(10001) = R(r)+1$.

A judgement in the depth system has the shape $R; \Gamma \vdash^{\delta} P$ and it should be interpreted as follows: the free variables of $!^{\delta} P$ may only occur at the depth specified by the context Γ, where depths are computed according to R. The inference rules of the depth system are presented in Table 3.1. We comment on the rules. The variable rule says that the current depth of a free variable is specified by the context. A region and the constant $*$ may appear at any depth. The λ-abstraction rule requires that the occurrence of x in M is at the same depth as the formal parameter; moreover it occurs at most once so that no duplication is possible at the current depth (Property (3)). The application rule says that we may only apply a term to another one if they are at the same depth. The $\mathsf{let}\,!$ rule requires that the bound occurrences of x are one level deeper than the current depth; note that there is no restriction on the number of occurrences of x since duplication would happen one level deeper than the current depth. The bang rule is better explained in a bottom-up way: crossing a modal occurrence increases the current depth by one. The key cases are those of read and write: the depth of these two operations is specified by the region context. The current depth of a store is always 0, however, the depth of the value in the store is specified by R (note that it corresponds to the revised definition of depth). We remark that R is constant in a judgement derivation.

$$\frac{}{R;\Gamma,x:\delta\vdash^{\delta} x}\qquad\frac{}{R;\Gamma\vdash^{\delta} r}\qquad\frac{}{R;\Gamma\vdash^{\delta} *}$$

$$\frac{\mathsf{FO}(x,M)\le 1\quad R;\Gamma,x:\delta\vdash^{\delta} M}{R;\Gamma\vdash^{\delta}\lambda x.M}\qquad\frac{R;\Gamma\vdash^{\delta} M_i\quad i=1,2}{R;\Gamma\vdash^{\delta} M_1M_2}$$

$$\frac{R;\Gamma\vdash^{\delta+1} M}{R;\Gamma\vdash^{\delta} !M}\qquad\frac{R;\Gamma\vdash^{\delta} M_1\quad R;\Gamma,x:(\delta+1)\vdash^{\delta} M_2}{R;\Gamma\vdash^{\delta}\mathsf{let}\ !x=M_1\ \mathsf{in}\ M_2}$$

$$\frac{}{R,r:\delta;\Gamma\vdash^{\delta}\mathsf{get}(r)}\qquad\frac{R,r:\delta;\Gamma\vdash^{\delta} V}{R,r:\delta;\Gamma\vdash^{\delta}\mathsf{set}(r,V)}$$

$$\frac{R,r:\delta;\Gamma\vdash^{\delta} V}{R,r:\delta;\Gamma\vdash^{0}(r\leftarrow V)}\qquad\frac{R;\Gamma\vdash^{\delta} P_i\quad i=1,2}{R;\Gamma\vdash^{\delta}(P_1\mid P_2)}$$

Table 3.1. Depth system for programs: $\lambda^{!\mathsf{R}}_{\delta}$

Definition 3 (well-formedness). *A program P is* well-formed *if for some R, Γ, δ a judgement $R;\Gamma\vdash^{\delta} P$ can be derived.*

Example 2. The program of Figure 2.1 is well-formed with the following derivation where $R(r)=0$:

$$\frac{\dfrac{R;\Gamma\vdash^{0}\mathsf{get}(r)\qquad\dfrac{\dfrac{R;\Gamma,x:1\vdash^{1} x}{R;\Gamma,x:1\vdash^{0} !x}}{R;\Gamma,x:1\vdash^{0}\mathsf{set}(r,!x)}}{R;\Gamma\vdash^{0}\mathsf{let}\ !x=\mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r,!x)}\qquad\dfrac{\vdots}{R;\Gamma\vdash^{0}(r\leftarrow !(\lambda x.x*))}}{R;\Gamma\vdash^{0}\mathsf{let}\ !x=\mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r,!x)\mid(r\leftarrow !(\lambda x.x*))}$$

On the other hand, the following term is not well-formed: $P=\lambda x.\mathsf{let}\ !y=x\ \mathsf{in}\ !(y!(yz))$. Indeed, the second occurrence of y in $!(y!(yz))$ is one level too deep, hence reduction may increase the depth by one. For example, $P!!V$ of depth 2 reduces to $!(!V!(!V)z)$ of depth 3.

We reconsider the troublesome programs with side effects. Program (A) is well-formed with judgement (i):

$$\begin{array}{lll}
R;\Gamma\vdash^{0} E[\mathsf{set}(r,!V)] & \text{with } R=r:\delta & (i)\\
R;\Gamma\vdash^{0} !M_r\mid(r\leftarrow !(\lambda y.M_{r'}))\mid(r'\leftarrow !(\lambda y.*)) & \text{with } R=r:1,r':2 & (ii)
\end{array}$$

Indeed, the occurrence $!V$ is now preserved at depth δ in the store. Program (B) is not well-formed since the read operation requires $R(r)=1$ and the write operations require $R(r)=0$. Program (C) is well-formed with judgement (ii); indeed its depth does not increase anymore because $!M_r$ has depth 2 but since $R(r)=1$ and $R(r')=2$, $(r\leftarrow !(\lambda y.M_{r'}))$ has depth 3 and $(r'\leftarrow !(\lambda y.*))$ has depth 2. Hence program (C) has already depth 3. Finally, it is worth noticing that the diverging program (D) is not well-formed since $\mathsf{get}(r)$ appears at depth 1 in $!M_r$ and at depth 2 in the store.

Theorem 1 (properties on the depth system). *The following properties hold:*

1. *If $R; \Gamma \vdash^{\delta} M$ and x occurs free in M then $x : \delta'$ belongs to Γ and all occurrences of x in $!^{\delta} M$ are at depth δ'.*
2. *If $R; \Gamma \vdash^{\delta} P$ then $R; \Gamma, \Gamma' \vdash^{\delta} P$.*
3. *If $R; \Gamma, x : \delta' \vdash^{\delta} M$ and $R; \Gamma \vdash^{\delta'} V$ then $R; \Gamma \vdash^{\delta} M[V/x]$ and $d(!^{\delta} M[V/x]) \leq max(d(!^{\delta} M), d(!^{\delta'} V))$.*
4. *If $R; \Gamma \vdash^{0} P$ and $P \rightarrow P'$ then $R; \Gamma \vdash^{0} P'$ and $d(P) \geq d(P')$.*

4 Elementary Bound

In this section, we prove that well-formed programs terminate in elementary time. To this end, we define a measure on programs based on the number of occurrences at each depth.

Definition 4 (measure). *Given a program P and $0 \leq i \leq d(M)$, let $\omega_i(P)$ be the number of occurrences in P of depth i increased by 2 (so $\omega_i(P) \geq 2$). We define $\mu_n^i(P)$ for $n \geq i \geq 0$ as follows:*

$$\mu_n^i(P) = (\omega_n(P), \ldots, \omega_{i+1}(P), \omega_i(P))$$

We write $\mu_n(P)$ for $\mu_n^0(P)$. We order the vectors of $n+1$ natural number with the (well-founded) lexicographic order $>$ from right to left.

To simplify the proofs of the following properties, we assume the occurrences labelled with $\mid$ and $r \leftarrow$ do not count in the measure and that $\mathsf{set}(r)$ counts for two occurrences, such that the measure strictly decreases on the rule $E[\mathsf{set}(r, V)] \rightarrow E[*] \mid (r \leftarrow V)$.

Following this assumption, we derive a termination property by observing that the measure strictly decreases during reduction.

Proposition 1 (termination). *If P is well-formed, $P \rightarrow P'$ and $n \geq d(P)$ then $\mu_n(P) > \mu_n(P')$.*

Proof. By a case analysis on the reduction rules. The crucial cases are those that increase the number of occurrences, namely both $\mathsf{let}\,!$ reductions: the one that is functional and the one that copies from the store. Thanks to the design of our depth system, we observe that both rules generate duplication of occurrences in the same way, hence we may only consider the functional case as an illustration where $P = E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M] \rightarrow P' = E[M[V/x]]$.

Let the occurrence of the redex $\mathsf{let}\ !x = !V\ \mathsf{in}\ M$ be at depth i. The restrictions on the formation of terms require that x may only occur in M at depth 1 and hence in P at depth $i+1$. We have that $\omega_i(P') = \omega_i(P) - 2$ because the $\mathsf{let}\,!$ node disappears. Clearly, $\omega_j(P) = \omega_j(P')$ if $j < i$. The number of occurrences of x in M is bounded by $k = \omega_{i+1}(P) \geq 2$. Thus if $j > i$ then $\omega_j(P') \leq k \cdot \omega_j(P)$.

Let's write, for $0 \leq i \leq n$, $\mu_n^i(P) \cdot k = (\omega_n(P) \cdot k, \omega_{n-1}(P) \cdot k, \ldots, \omega_i(P) \cdot k)$. Then we have:

$$\mu_n(P') \leq (\mu_n^{i+1}(P) \cdot k, \omega_i(P) - 2, \mu_{i-1}(P)) \tag{4.1}$$

and finally $\mu_n(P) > \mu_n(P')$.

We now want to show that termination is actually in elementary time. We recall that a function f on integers is elementary if there exists a k such that for any n, $f(n)$ can be computed in time $\mathcal{O}(t(n,k))$ where:

$$t(n,0) = 2^n, \qquad t(n,k+1) = 2^{t(n,k)} \ .$$

Definition 5 (tower functions). *We define a family of tower functions $t_\alpha(x_1, \ldots, x_n)$ by induction on n where we assume $\alpha \geq 1$ and $x_i \geq 2$:*

$$\begin{aligned} t_\alpha() &= 0 \\ t_\alpha(x_1, x_2, \ldots, x_n) &= (\alpha \cdot x_1)^{2^{t_\alpha(x_2,\ldots,x_n)}} \quad n \geq 1 \end{aligned}$$

Then we need to prove the following crucial lemma.

Lemma 1 (shift). *Assuming $\alpha \geq 1$ and $\beta \geq 2$, the following property holds for the tower functions with $x, \mathbf{x}$ ranging over numbers greater or equal to 2:*

$$t_\alpha(\beta \cdot x, x', \mathbf{x}) \leq t_\alpha(x, \beta \cdot x', \mathbf{x})$$

Now, by a closer look at the shape of the lexicographic ordering during reduction, we are able to compose the decreasing measure with a tower function.

Theorem 2 (elementary bound). *Let P be a well-formed program with $\alpha = d(P)$ and let t_α denote the tower function with $\alpha+1$ arguments. Then if $P \to P'$ then $t_\alpha(\mu_\alpha(P)) > t_\alpha(\mu_\alpha(P'))$.*

Proof. We exemplify the proof for $\alpha = 2$ and the crucial case where

$$P = \mathsf{let}\ !x = !V \ \mathsf{in}\ M \to P' = M[V/x]$$

Let $\mu_2(P) = (x, y, z)$ such that $x = \omega_2(P)$, $y = \omega_1(P)$ and $z = \omega_0(P)$. We want to show that: $t_2(\mu_2(P')) < t_2(\mu_2(P))$. We have:

$$\begin{aligned} t_2(\mu_2(P')) &\leq t_2(x \cdot y, y \cdot y, z-2) && \text{by inequality (4.1)} \\ &\leq t_2(x, y^3, z-2) && \text{by Lemma 1} \end{aligned}$$

Hence we are left to show that: $t_2(y^3, z-2) < t_2(y,z)$, *i.e.* , $(2y^3)^{2^{2(z-2)}} < (2y)^{2^{2z}}$. We have: $(2y^3)^{2^{2(z-2)}} \leq (2y)^{3 \cdot 2^{2(z-2)}}$. Thus we need to show: $3 \cdot 2^{2(z-2)} < 2^{2z}$ which is true. Hence $t_2(\mu_2(P')) < t_2(\mu_2(P))$.

This shows that the number of reduction steps of a program P is bounded by an elementary function where the height of the tower only depends on $d(P)$. We also note that if $P \xrightarrow{*} P'$ then $t_\alpha(\mu_\alpha(P))$ bounds the size of P'. Thus we can conclude with the following corollary.

Corollary 1. *The normalisation of programs of bounded depth can be performed in time elementary in the size of the terms.*

We remark that if P is a purely functional term then the elementary bound holds under an arbitrary reduction strategy.

5 An Elementary Affine Type System

The depth system entails termination in elementary time but does *not* guarantee that programs ‘do not go wrong’. In particular, the introduction and elimination of bangs during evaluation may generate programs that deadlock, *e.g.*,

$$\mathsf{let}\ !y = (\lambda x.x)\ \mathsf{in}\ !(yy) \tag{5.1}$$

is well-formed but the evaluation is stuck. In this section we introduce an elementary affine type system ($\lambda^{!\mathsf{R}}_{EA}$) that guarantees that programs cannot deadlock (except when trying to read an empty store).

The upper part of Table 5.1 introduces the syntax of types and contexts. Types are denoted with $\alpha, \alpha', \ldots$. Note that we distinguish a special behaviour type $\mathbf{B}$ which is given to the entities of the language which are not supposed to return a value (such as a store or several terms in parallel) while types of entities that may return a value are denoted with A. Among the types A, we distinguish type variables $t, t', \ldots$, a terminal type $\mathbf{1}$, an affine functional type $A \multimap \alpha$, the type $!A$ of terms of type A that can be duplicated, the type $\forall t.A$ of polymorphic terms and the type $\mathsf{Reg}_r A$ of the region r containing values of type A. Hereby types may depend on regions.

$$\begin{array}{ll}
t, t', \ldots & \text{(Type variables)} \\
\alpha ::= \mathbf{B} \mid A & \text{(Types)} \\
A ::= t \mid \mathbf{1} \mid A \multimap \alpha \mid !A \mid \forall t.A \mid \mathsf{Reg}_r A & \text{(Value-types)} \\
\Gamma ::= x_1 : (\delta_1, A_1), \ldots, x_n : (\delta_n, A_n) & \text{(Variable contexts)} \\
R ::= r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n) & \text{(Region contexts)}
\end{array}$$

$$\frac{}{R \downarrow t} \qquad \frac{}{R \downarrow \mathbf{1}} \qquad \frac{}{R \downarrow \mathbf{B}} \qquad \frac{R \downarrow A \quad R \downarrow \alpha}{R \downarrow (A \multimap \alpha)}$$

$$\frac{R \downarrow A}{R \downarrow !A} \qquad \frac{r : (\delta, A) \in R}{R \downarrow \mathsf{Reg}_r A} \qquad \frac{R \downarrow A \quad t \notin R}{R \downarrow \forall t.A}$$

$$\frac{\forall r : (\delta, A) \in R \quad R \downarrow A}{R \vdash} \qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha} \qquad \frac{\forall x : (\delta, A) \in \Gamma \quad R \vdash A}{R \vdash \Gamma}$$

Table 5.1. Types and contexts

In contexts, natural numbers δ_i play the same role as in the depth system. Writing $x : (\delta, A)$ means that the variable x ranges on values of type A and may occur at depth δ. Writing $r : (\delta, A)$ means that addresses related to region r contain values of type A and that read and writes on r may only happen at depth δ. The typing system will additionally guarantee that whenever we use a type $\mathsf{Reg}_r A$ the region context contains an hypothesis $r : (\delta, A)$.

Because types depend on regions, we have to be careful in stating in Table 5.1 when a region-context and a type are compatible ($R \downarrow \alpha$), when a region context is well-formed ($R \vdash$), when a type is well-formed in a region context ($R \vdash \alpha$) and when a context is well-formed in a region context ($R \vdash \Gamma$). A more informal way to express the condition is to say that a judgement $r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n) \vdash \alpha$ is well formed provided that: (1) all the region names occurring in the types $A_1, \ldots, A_n, \alpha$ belong to the set $\{r_1, \ldots, r_n\}$, (2) all types of the shape $\mathsf{Reg}_{r_i} B$ with $i \in \{1, \ldots, n\}$ and occurring in the types $A_1, \ldots, A_n, \alpha$ are such that $B = A_i$. We notice the following substitution property on types.

Proposition 2. *If $R \vdash \forall t.A$ and $R \vdash B$ then $R \vdash A[B/t]$.*

Example 3. One may verify that $r : (\delta, \mathbf{1} \multimap \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$ can be derived while the following judgements cannot: $r : (\delta, \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$, $r : (\delta, \mathsf{Reg}_r\mathbf{1}) \vdash \mathbf{1}$.

A typing judgement takes the form: $R; \Gamma \vdash^\delta P : \alpha$. It attributes a type α to the program P at depth δ, in the region context R and the context Γ. Table 5.2 introduces an elementary affine type system *with regions*. One can see that the δ's are treated as in the depth system. Note that a region r may occur at any depth. In the $\mathsf{let}\,!$ rule, M should be of type $!A$ since x of type A appears

$$\frac{R \vdash \Gamma \quad x : (\delta, A) \in \Gamma}{R; \Gamma \vdash^\delta x : A} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash^\delta * : \mathbf{1}} \qquad \frac{R \vdash \Gamma \quad r : (\delta', A) \in R}{R; \Gamma \vdash^\delta r : \mathsf{Reg}_r A}$$

$$\frac{\begin{array}{c}\mathsf{FO}(x, M) \leq 1 \\ R; \Gamma, x : (\delta, A) \vdash^\delta M : \alpha\end{array}}{R; \Gamma \vdash^\delta \lambda x.M : A \multimap \alpha} \qquad \frac{R; \Gamma \vdash^\delta M : A \multimap \alpha \quad R; \Gamma \vdash^\delta N : A}{R; \Gamma \vdash^\delta MN : \alpha}$$

$$\frac{R; \Gamma \vdash^{\delta+1} M : A}{R; \Gamma \vdash^\delta !M : !A} \qquad \frac{R; \Gamma \vdash^\delta M : !A \quad R; \Gamma, x : (\delta+1, A) \vdash^\delta N : B}{R; \Gamma \vdash^\delta \mathsf{let}\ !x = M\ \mathsf{in}\ N : B}$$

$$\frac{R; \Gamma \vdash^\delta M : A \quad t \notin (R; \Gamma)}{R; \Gamma \vdash^\delta M : \forall t.A} \qquad \frac{R; \Gamma \vdash^\delta M : \forall t.A \quad R \vdash B}{R; \Gamma \vdash^\delta M : A[B/t]}$$

$$\frac{r : (\delta, A) \in R \quad R \vdash \Gamma}{R; \Gamma \vdash^\delta \mathsf{get}(r) : A} \qquad \frac{\begin{array}{c}r : (\delta, A) \in R \\ R; \Gamma \vdash^\delta V : A\end{array}}{R; \Gamma \vdash^\delta \mathsf{set}(r, V) : \mathbf{1}} \qquad \frac{\begin{array}{c}r : (\delta, A) \in R \\ R; \Gamma \vdash^\delta V : A\end{array}}{R; \Gamma \vdash^0 (r \leftarrow V) : \mathbf{B}}$$

$$\frac{R; \Gamma \vdash^\delta P : \alpha \quad R; \Gamma \vdash^\delta S : \mathbf{B}}{R; \Gamma \vdash^\delta (P \mid S) : \alpha} \qquad \frac{P_i \text{ not a store } i = 1,2 \quad R; \Gamma \vdash^\delta P_i : \alpha_i}{R; \Gamma \vdash^\delta (P_1 \mid P_2) : \mathbf{B}}$$

Table 5.2. An elementary affine type system: $\lambda^{\mathsf{!R}}_{EA}$

one level deeper. A program in parallel with a store should have the type of the program since we might be interested in the value the program reduces to; however, two programs in parallel cannot reduce to a single value, hence we give them a behaviour type. The polymorphic rules are straightforward where $t \notin (R; \Gamma)$ means t does not occur free in a type of R or Γ.

Example 4. The well-formed program (C) can be given the following typing judgement: $R; \_ \vdash^0 !(M_r) \mid (r \leftarrow !(\lambda y.M_{r'})) \mid (r' \leftarrow !(\lambda y.*)) : !!\mathbf{1}$ where: $R = r : (1, !(\mathbf{1} \multimap \mathbf{1})), r' : (2, !(\mathbf{1} \multimap \mathbf{1}))$. Also, we remark that the deadlocking program (5.1) admits no typing derivation.

Theorem 3 (subject reduction and progress). *The following properties hold.*

1. (Well-formedness) *Well-typed programs are well-formed.*
2. (Weakening) *If $R; \Gamma \vdash P : \alpha$ and $R \vdash^\delta \Gamma, \Gamma'$ then $R; \Gamma, \Gamma' \vdash^\delta P : \alpha$.*
3. (Substitution) *If $R; \Gamma, x : (\delta', A) \vdash^\delta M : \alpha$ and $R; \Gamma' \vdash^{\delta'} V : A$ and $R \vdash \Gamma, \Gamma'$ then $R; \Gamma, \Gamma' \vdash^\delta M[V/x] : \alpha$.*
4. (Subject Reduction) *If $R; \Gamma \vdash^\delta P : \alpha$ and $P \to P'$ then $R; \Gamma \vdash^\delta P' : \alpha$.*
5. (Progress) *Suppose P is a closed typable program which cannot reduce. Then P is structurally equivalent to a program*

$$M_1 \mid \cdots \mid M_m \mid S_1 \mid \cdots \mid S_n \quad m, n \geq 0$$

where M_i is either a value or can be decomposed as a term $E[\mathsf{get}(r)]$ such that no value is associated with the region r in the stores $S_1, \ldots, S_n$.

6 Expressivity

In this section, we consider two results that illustrate the expressivity of the elementary affine type system. First we show that all elementary functions can be represented and second we develop an example of iterative program with side effects.

Completeness The representation result just relies on the functional core of the language $\lambda^!_{EA}$. Building on the standard concept of Church numeral, Table 6.1 provides a representation for natural numbers and the multiplication function. We denote with $\mathbb{N}$ the set of natural numbers. The precise notion of representation is spelled out in the following definitions where by strong β-reduction we mean that reduction under λ's is allowed.

Definition 6 (number representation). *Let $\emptyset \vdash^\delta M : \mathsf{N}$. We say M* represents *$n \in \mathbb{N}$, written $M \Vdash n$, if, by using a strong β-reduction relation, $M \xrightarrow{*} \overline{n}$.*

Definition 7 (function representation). *Let $\emptyset \vdash^\delta F : (\mathsf{N}_1 \multimap \ldots \multimap \mathsf{N}_k) \multimap !^p\mathsf{N}$ where $p \geq 0$ and $f : \mathbb{N}^k \to \mathbb{N}$. We say F* represents *f, written $F \Vdash f$, if for all M_i and $n_i \in \mathbb{N}$ where $1 \leq i \leq k$ such that $\emptyset \vdash^\delta M_i : N$ and $M_i \Vdash n_i$, $FM_1 \ldots M_k \Vdash f(n_1, \ldots, n_k)$.*

$$\begin{array}{rcll} \mathsf{N} & = & \forall t.!(t \multimap t) \multimap !(t \multimap t) & \text{(type of numerals)} \\ \\ \overline{n} & : & \mathsf{N} & \text{(numerals)} \\ \overline{n} & = & \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.f(\cdots(fx)\cdots)) & \\ \\ \mathsf{mult} & : & \mathsf{N} \multimap (\mathsf{N} \multimap \mathsf{N}) & \text{(multiplication)} \\ \mathsf{mult} & = & \lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ n(m!f) & \end{array}$$

Table 6.1. Representation of natural numbers and the multiplication function

Elementary functions are characterized as the smallest class of functions containing zero, successor, projection, subtraction and which is closed by composition and bounded summation/product. These functions can be represented in the sense of Definition 7 by adapting the proofs from Danos and Joinet [10].

Theorem 4 (completeness). *Every elementary function is representable in* $\lambda^!_{EA}$.

Iteration with Side Effects We rely on a slightly modified language where reads, writes and stores relate to concrete addresses rather than to abstract regions. In particular, we introduce terms of the form $\nu x\ M$ to generate a fresh address name x whose scope is M. One can then write the following program:

$$\nu x\ ((\lambda y.\mathsf{set}(y, V))x) \xrightarrow{*} \nu x\ * \mid (x \leftarrow V)$$

where x and y relate to a region r, *i.e.* they are of type $\mathsf{Reg}_r A$. Our type system can be easily adapted by associating region types with the address names. Next we show that it is possible to program the iteration of operations producing a side effect on an inductive data structure. Specifically, in the following we show how to iterate, possibly in parallel, an update operation on a list of addresses of the store. The examples have been tested on a running implementation of the language.

Following Church encodings, we define the representation of lists and the associated iterator in Table 6.2. Here is the function multiplying the numeral

$$\begin{array}{rcll} \mathsf{List}\ A & = & \forall t.!(A \multimap t \multimap t) \multimap !(t \multimap t) & \text{(type of lists)} \\ \\ [u_1, \ldots, u_n] & : & \mathsf{List}\ A & \text{(list represent.)} \\ [u_1, \ldots, u_n] & = & \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.fu_1(fu_2 \ldots (fu_n x)) & \\ \\ \mathsf{list\_it} & : & \forall u.\forall t.!(u \multimap t \multimap t) \multimap \mathsf{List}\ u \multimap !t \multimap !t & \text{(iterator on lists)} \\ \mathsf{list\_it} & = & \lambda f.\lambda l.\lambda z.\mathsf{let}\ !z = z\ \mathsf{in}\ \mathsf{let}\ !y = lf\ \mathsf{in}\ !(yz) & \end{array}$$

Table 6.2. Representation of lists

pointed by an address at region r:

$$\begin{aligned} &\mathsf{update}\ :\ !\mathsf{Reg}_r\mathsf{N} \multimap !\mathbf{1} \multimap !\mathbf{1} \\ &\mathsf{update} = \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ \lambda z.!((\lambda y.\mathsf{set}(x,y))(\mathsf{mult}\ \overline{2}\ \mathsf{get}(x))) \end{aligned}$$

Consider the following list of addresses and stores:

$$[!x, !y, !z] \mid (x \leftarrow \overline{m}) \mid (y \leftarrow \overline{n}) \mid (z \leftarrow \overline{p})$$

Note that the bang constructors are needed to match the type $!\mathsf{Reg}_r\mathsf{N}$ of the argument of update. Then we define the iteration as:

$$\mathsf{run} : !!\mathbf{1} \qquad \mathsf{run} = \mathsf{list\_it}\ !\mathsf{update}\ [!x, !y, !z]\ !!*$$

Notice that it is well-typed with $R = r : (2, \mathsf{N})$ since both the read and the write appear at depth 2. Finally, the program reduces by updating the store as expected:

$$\mathsf{run} \mid (x \leftarrow \overline{m}) \mid (y \leftarrow \overline{n}) \mid (z \leftarrow \overline{p}) \quad \overset{*}{\to} \quad !!\mathbf{1} \mid (x \leftarrow \overline{2m}) \mid (y \leftarrow \overline{2n}) \mid (z \leftarrow \overline{2p})$$

Building on this example, suppose we want to write a program with three concurrent threads where each thread multiplies by 2 the memory cells pointed by a list. Here is a function waiting to apply a functional f to a value x in three concurrent threads:

$$\begin{aligned} &\mathsf{gen\_threads}\ :\ \forall t.\forall t'.!(t \multimap t') \multimap !t \multimap \mathbf{B} \\ &\mathsf{gen\_threads} = \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ !(fx) \mid !(fx) \mid !(fx) \end{aligned}$$

We define the functional F as run but parametric in the list:

$$F : \mathsf{List}\ !\mathsf{Reg}_r\mathsf{N} \multimap !!\mathbf{1} \qquad F = \lambda l.\mathsf{list\_it}\ !\mathsf{update}\ l\ !!*$$

And the final term is simply:

$$\mathsf{run\_threads} : \mathbf{B} \qquad \mathsf{run\_threads} = \mathsf{gen\_threads}\ !F\ ![!x, !y, !z]$$

where $R = r : (3, !\mathsf{N})$. Our program then reduces as follows:

$$\begin{aligned} &\mathsf{run\_threads} &&\mid (x \leftarrow \overline{m}) &&\mid (y \leftarrow \overline{n}) &&\mid (z \leftarrow \overline{p}) \\ \overset{*}{\to}\ &!!!\mathbf{1} \mid !!!\mathbf{1} \mid !!!\mathbf{1} &&\mid (x \leftarrow \overline{8m}) &&\mid (y \leftarrow \overline{8n}) &&\mid (z \leftarrow \overline{8p}) \end{aligned}$$

Note that different thread interleavings are possible but in this particular case the reduction is confluent.

7 Conclusion

We have introduced a type system for a higher-order functional language with multithreading and side effects that guarantees termination in elementary time thus providing a significant extension of previous work that had focused on purely functional programs.

In the proposed approach, the depth system plays a key role and allows for a relatively simple presentation. In particular we notice that we can dispense with the notion of *stratified region* that arises in recent work on the termination of higher-order programs with side effects [1,7]. Hence, it becomes possible to type circular stores in $\lambda^{!\mathsf{R}}_{EA}$ like *e.g.* $(r \leftarrow \lambda x.\mathsf{set}(r,x); \mathsf{get}(r))$ whereas the stratification condition precludes them. Note that this store is going to be consumed when r is read. However, concerning duplicable stores (*i.e.* of the shape $(r \leftarrow !V)$), the value V is implicitely stratified by the difference of depth between V and r. We note that we can also dispense with the distinction between affine and intuitionistic hypotheses [6,2].

As a future work, we would like to adapt our approach to polynomial time. In another direction, one could ask if it is possible to program in a simplified language without bangs and then try to infer types or depths.

Acknowledgements We would like to thank Patrick Baillot for numerous helpful discussions and a careful reading on a draft version of this paper, plus the anonymous referees for their feedbacks.

References

1. R. M. Amadio. On stratified regions. In *APLAS'09*, volume 5904 of *LNCS*, pages 210–225. Springer, 2009.
2. R. M. Amadio, P. Baillot, and A. Madet. An affine-intuitionistic system of types and effects: confluence and termination. Technical report, Laboratoire PPS, 2009. `http://hal.archives-ouvertes.fr/hal-00438101/`.
3. A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM Trans. Comput. Log.*, 3(1):137–175, 2002.
4. P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In *ESOP'10*, volume 6012 of *LNCS*, pages 104–124. Springer, 2010.
5. P. Baillot and K. Terui. A feasible algorithm for typing in elementary affine logic. In *TLCA'05*, volume 3461 of *LNCS*, pages 55–70. Springer, 2005.
6. A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, The Laboratory for Foundations of Computer Science, University of Edinburgh, 1996.
7. G. Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208(6):716–736, 2010.
8. P. Coppola, U. Dal Lago, and S. Ronchi Della Rocca. Light logics and the call-by-value lambda calculus. *Logical Methods in Computer Science*, 4(4), 2008.
9. P. Coppola and S. Martini. Optimizing optimal reduction: A type inference algorithm for elementary affine logic. *ACM Trans. Comput. Log.*, 7:219–260, 2006.
10. V. Danos and J.-B. Joinet. Linear logic and elementary time. *Inf. Comput.*, 183(1):123 – 137, 2003.
11. J.-Y. Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.
12. U. D. Lago, S. Martini, and D. Sangiorgi. Light logics and higher-order processes. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 46–60, 2010.
13. A. Madet and R. M. Amadio. Elementary affine λ-calculus with multithreading and side effects. Technical report, Laboratoire PPS, 2011. `http://hal.archives-ouvertes.fr/hal-00569095/`.
14. K. Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3-4):253–280, 2007.