A polynomial time λ-calculus with multithreading and side effects

Antoine Madet

A Polynomial Time λ-calculus with Multithreading and Side Effects *

Antoine Madet

Univ Paris Diderot, Sorbonne Paris Cité
PPS, UMR 7126, CNRS, F-75205 Paris, France
madet@pps.univ-paris-diderot.fr

Abstract

The framework of *light logics* has been extensively studied to control the complexity of higher-order functional programs. We propose an extension of this framework to multithreaded programs with side effects, focusing on the case of polynomial time. After introducing a modal λ-calculus with parallel composition and *regions*, we prove that a realistic call-by-value evaluation strategy can be computed in polynomial time for a class of well-formed programs. The result relies on the simulation of call-by-value by a polynomial *shallow-first* strategy which preserves the evaluation order of side effects. Then, we provide a polynomial type system that guarantees that well-typed programs do not go wrong. Finally, we illustrate the expressivity of the type system by giving a programming example of concurrent iteration producing side effects over an inductive data structure.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Formal Definitions and Theory; F.2 [*Analysis of Algorithms and Problem Complexity*]: General

Keywords λ-calculus, side effect, region, thread, resource analysis.

1. Introduction

Quantitative resource analysis of programs is a challenging task in computer science. Besides being essential for the development of safety-critical systems, it provides interesting viewpoints on the structure of programs.

The framework of *light logics* (see *e.g.* **LLL** [12], **ELL** [10], **SLL** [13]) which originates from Linear Logic [11], have been deeply studied to control the complexity of higher-order functional programs. In particular, polynomial time λ-calculi [5, 18] have been proposed as well as various type systems [8, 9] guaranteeing complexity bounds of functional programs. Recently, Amadio and the author proposed an extension of the framework to a higher-order functional language with multithreading and side effects [16], focusing on the case of elementary time (**ELL**).

In this paper, we consider a more reasonable complexity class: polynomial time. The functional core of the language is the *light* λ-calculus [18] that features the modalities *bang* (written '!') and *paragraph* (written '§') of **LLL**. The notion of *depth* (the number of nested modalities) which is standard in light logics is used to control the duplication of data during the execution of programs. The language is extended with side effects by means of read and write operations on *regions* which were introduced to represent areas of the store [15]. Threads can be put in parallel and interact through a shared state.

There appears to be no direct combinatorial argument to bound a call-by-value evaluation strategy by a polynomial. However, the *shallow-first* strategy (*i.e.* redexes are eliminated in a depth-increasing order) is known to be polynomial in the functional case [4, 12]. Using this result, Terui shows [18] that a class of *well-formed* light λ-terms strongly terminates in polynomial time (*i.e.* every reduction strategy is polynomial) by proving that any reduction sequence can be simulated by a *longer* one which is shallow-first. Following this method, our contribution is to show that a class of well-formed call-by-value programs with side effects and multithreading can be simulated in polynomial time by shallow-first reductions. The bound covers any scheduling policy and takes thread generation into account.

Reordering a reduction sequence into a shallow-first one is non-trivial: the evaluation order of side effects must be kept unchanged in order to preserve the semantics of the program. An additional difficulty is that reordering produces non call-by-value sequences but fails for an arbitrary larger relation (which may even require exponential time). We identify an intermediate *outer-bang* relation $\longrightarrow_{\mathsf{ob}}$ which can be simulated by shallow-first ordering and this allows us to simulate the call-by-value relation $\longrightarrow_{\mathsf{v}}$ which is contained in the outer-bang relation. We illustrate this development in Figure 1.

The paper is organized as follows. We start by presenting the language with multithreading and regions in Section 2 and define the largest reduction relation. Then, we introduce a *polynomial depth system* in Section 3 to control the depth of program occurrences. Well-formed programs in the depth system follow Terui's discipline [18] on the functional side and the *stratification of regions* by depth level that we introduced previously [16]. We prove in Section 4 that the class of outer-bang strategies (containing call-by-value) can be simulated by shallow-first reductions of exactly the same length. We review the proof of polynomial soundness of the shallow-first strategy in Section 5. We provide a *polynomial type system* in Section 6 which results from a simple decoration of the polynomial depth system with linear types. We derive the stan-

* 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).

PPDP'12, September 19–21, 2012, Leuven, Belgium.

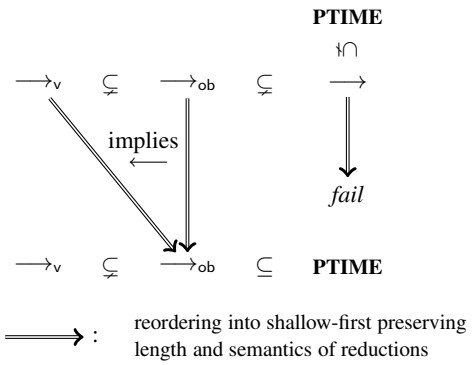

Figure 1. Simulation by shallow-first ordering

dard subject reduction proposition and progress proposition which states that well-types programs reduce to values. Finally, we illustrate the expressivity of the type system in Section 7 by showing that it is polynomially complete in the extensional sense and we give a programming example of a concurrent iteration producing side effects over an inductive data structure.

2. A modal λ-calculus with multithreading and regions

As mentioned previously, the functional core of the language is a modal λ-calculus with constructors and destructors for the modalities '!' and '§' that are used to control the duplication of data. The global store is partitioned into a finite number of regions where each region abstracts a set of memory locations. Following [1], side effects are produced by read and write operators on regions. A parallel operator allows to evaluate concurrently several terms which can communicate through regions. As we shall see in Section 7, this abstract non-deterministic language entails complexity bounds for languages with concrete memory locations representing *e.g.* references, channels or signals.

The syntax of the language is presented in Figure 2. We have

-variables	$x, y, \ldots$		
-regions	$r, r', \ldots$		
-terms	M	$::=$	$x \mid r \mid \star \mid \lambda x.M \mid MM \mid !M \mid \S M$
			$\mathsf{let}\ !x = M\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = M\ \mathsf{in}\ M$
			$\mathsf{get}(r) \mid \mathsf{set}(r, M) \mid (M \parallel M)$
-stores	S	$::=$	$r \Leftarrow M \mid (S \parallel S)$
-programs	P	$::=$	$M \mid S \mid (P \parallel P)$

Figure 2. Syntax of the language

the usual set of variables $x, y, \ldots$ and a set of regions $r, r', \ldots$ The set of terms M contains variables, regions, the terminal value (unit) $\star$, λ-abstractions, applications, modal terms $!M$ and $\S M$ (resp. called !-terms and §-terms) and the associated let !-binders and let §-binders. We have an operator $\mathsf{get}(r)$ to read a region r, an operator $\mathsf{set}(r, M)$ to assign a term M to a region r and a parallel operator $(M \parallel N)$ to evaluate M and N in parallel. A store S is the composition of several assignments $r \Leftarrow M$ in parallel and a program P is the combination of several terms and stores in parallel. Note that stores are global, *i.e.* they always occur in empty contexts.

In the following we write † for $\dagger \in \{!, \S\}$ and we define $\dagger^0 M = M$ and $\dagger^{n+1} M = \dagger(\dagger^n M)$. Terms $\lambda x.M$ and $\mathsf{let}\ \dagger x = N\ \mathsf{in}\ M$ bind occurrences of x in M. 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 $\mathsf{FO}(x, M)$. The number of free occurrences in M is denoted by $\mathsf{FO}(M)$. $M[N/x]$ denotes the term M in which each free occurrence of x has been substituted by N.

Each program has an *abstract syntax tree* where variables, regions and unit constants are leaves, λ-abstractions and †-terms have one child, and applications and let †-binders have two children. An example is given in Figure 3. A path starting from the root to a

$$P = \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, (!x)(\S x)) \parallel r \Leftarrow\ !(\lambda x.x\star)$$

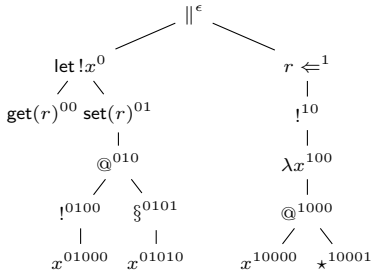

Figure 3. Syntax tree and addresses of P

node of the tree denotes an *occurrence* of the program whose address is a word $w \in \{0, 1\}^*$ hereby denoted in exponent form. We write $w \sqsubseteq w'$ when w is a prefix of w'. We denote the number of occurrences in P by $|P|$.

The operational semantics of the language is given in Figure 4. In order to prove the later simulation result, the largest reduction relation $\longrightarrow$ (which shall contain call-by-value) is presented.

-structural rules-

$$\begin{array}{rcl} P \parallel P' & \equiv & P' \parallel P \\ (P \parallel P') \parallel P'' & \equiv & P \parallel (P' \parallel P'') \end{array}$$

-evaluation contexts-

$$\begin{array}{rcl} E & ::= & [\cdot] \mid \lambda x.E \mid EM \mid ME \mid !E \mid \S E \\ & & \mathsf{let}\ !x = E\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = E\ \mathsf{in}\ M \\ & & \mathsf{let}\ !x = M\ \mathsf{in}\ E \mid \mathsf{let}\ \S x = M\ \mathsf{in}\ E \\ & & \mathsf{set}(r, E) \mid r \Leftarrow E \mid (E \parallel P) \mid (P \parallel E) \end{array}$$

-reduction rules-

$$\begin{array}{lrcl} (\beta) & E[(\lambda x.M)N] & \longrightarrow & E[M[N/x]] \\ (!) & E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] & \longrightarrow & E[M[N/x]] \\ (\S) & E[\mathsf{let}\ \S x = \S N\ \mathsf{in}\ M] & \longrightarrow & E[M[N/x]] \\ (\mathsf{get}) & E[\mathsf{get}(r)] \parallel r \Leftarrow M & \longrightarrow & E[M] \\ (\mathsf{set}) & E[\mathsf{set}(r, M)] & \longrightarrow & E[\star] \parallel r \Leftarrow M \text{–if } \mathsf{FV}(M) = \emptyset \\ (\mathsf{gc}) & E[\star \parallel M] & \longrightarrow & E[M] \end{array}$$

Figure 4. Operational semantics

Programs are considered up to a structural equivalence $\equiv$ which contains the equations for α-renaming, commutativity and associativity of parallel composition. Reduction rules apply modulo structural equivalence, in an evaluation context E which can be any program with exactly one occurrence of a special variable '$[\cdot]$', called the *hole*. We write $E[M]$ for $E[M/[\cdot]]$. Each rule is identified by its name. (β) is the usual β-reduction. $(\dagger)$ are rules for filtering modal terms. (get) is for *consuming* a term from a region. (set) is for *assigning* a closed term to a region. (gc) is for *erasing* a terminated thread.

First, note that the reduction rule (set) generates a *global* assignment, that is out of the evaluation context E. In turn, we require M

to be closed such that it does not contain variables bound in E. Second, several terms can be assigned to a single region. This cumulative semantics allows the simulation of several memory locations by a single region. In turn, reading a region consists in consuming *non-deterministically* one of the assigned terms.

The reduction is very 'liberal' with side effects. The contexts $(P \parallel E)$ and $(E \parallel P)$ embed any scheduling of threads. Moreover, contexts of the shape $r \Leftarrow E$ allow evaluation in the store as exemplified in the following possible reduction:

$$\begin{aligned}\mathsf{set}(r, \lambda x.\mathsf{get}(r)) \parallel r \Leftarrow M &\longrightarrow \star \parallel r \Leftarrow \lambda x.\mathsf{get}(r) \parallel r \Leftarrow M \\ &\longrightarrow \star \parallel r \Leftarrow \lambda x.M\end{aligned}$$

In the rules $(\beta), (\dagger), (\mathsf{gc})$, the *redex* denotes the term inside the context of the left hand-side and the *contractum* denotes the term inside the context of the right hand-side. In the rule (get), the redex is $\mathsf{get}(r)$ and the contractum is M. In the rule (set), the redex is $\mathsf{set}(r, M)$ and the contractum is M. Finally, $\longrightarrow^+$ denotes the transitive closure of $\longrightarrow$ and $\longrightarrow^*$ denotes the reflexive closure of $\longrightarrow^+$.

3. A polynomial depth system

In this section, we first review the principles of well-formed light λ-terms (Subsection 3.1) and then the stratification of regions by depth level (Subsection 3.2). Eventually we combine the two as a set of inference rules that characterizes a class of *well-formed* programs (Subsection 3.3).

3.1 On light λ-terms

First, we define the notion of depth.

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

With reference to Figure 3, $d(01000) = d(01010) = d(100) = d(1000) = d(10000) = d(10001) = 1$, whereas other occurrences have depth 0. In particular, $d(0100) = d(0101) = d(10) = 0$; what matters in computing the depth of an occurrence is the number of $\dagger$'s that precede strictly the end node. Thus $d(P) = 1$. In the sequel, we say that a program *occurs* at depth i when it corresponds to an occurrence of depth i. For example, $\mathsf{get}(r)$ occur at depth 0 in P. We write $\xrightarrow{i}$ when the redex occurs at depth i; we write $|P|_i$ for the number of occurrences at depth i of P.

Then we can define *shallow-first* reductions.

Definition 2. *A* shallow-first reduction sequence $P_1 \xrightarrow{i_1} P_2 \xrightarrow{i_2} \ldots \xrightarrow{i_n} P_n$ *is such that* $m < n$ *implies* $i_m \leq i_n$. *A* shallow-first strategy *is a strategy that produces shallow-first sequences.*

The polynomial soundness of shallow-first strategies relies on the following properties: when $P \overset{i}{\longrightarrow}{}^* P'$,

$$d(P') \leq d(P) \tag{3.1}$$

$$|P'|_j \leq |P|_j \text{ for } j < i \tag{3.2}$$

$$|P'|_i < |P|_i \tag{3.3}$$

$$|P'| \leq |P|^2 \tag{3.4}$$

To see this in a simple way, assume P is a program such that $d(P) = 2$. By properties (3.1),(3.2),(3.3) we can eliminate all the redexes of P with the shallow-first sequence $P \overset{0}{\longrightarrow}{}^* P' \overset{1}{\longrightarrow}{}^* P'' \overset{2}{\longrightarrow}{}^* P'''$. By property (3.4), $|P'''| \leq |P|^8$. By properties (3.3) the length l of the sequence is such that $l \leq |P| + |P'| + |P''| = p$. Since we can show that $p \leq |P|^8$ we conclude that the shallow-first evaluation of P can be computed in polynomial time.

The well-formedness criterions of light λ-terms are intended to ensure the above four properties. These criterions can be summarized as follows:

- λ-abstraction is affine: in $\lambda x.M$, x may occur at most once and at depth 0 in M.
- let !-binders are for duplication: in $\mathsf{let}\ !x = M\ \mathsf{in}\ N$, x may occur arbitrarily many times and at depth 1 in N.
- let §-binders are affine: in $\mathsf{let}\ \S x = M\ \mathsf{in}\ N$, x may occur at most once and at depth 1 in N. The depth of x must be due to a § modality.
- a !-term may contain at most one occurrence of free variable, whereas a §-term can contain many occurrences of free variables.

By the first three criterions, we observe the following. The depth of a term never increases (property (3.1)) since the reduction rules (β),(!) and (§) substitute a term for a variable occurring at the same depth. Reduction rules (β) and (§) are strictly size-decreasing since the corresponding binders are affine. A reduction (!) is strictly size-decreasing at the depth where the redex occurs but potentially size-increasing at deeper levels. Therefore properties (3.2) and (3.3) are also guaranteed. The fourth criterion is intended to ensure a quadratic size increase (property (3.4)). Indeed, take the term Z borrowed from [18] that respects the first three criterions but not the fourth:

$$\begin{aligned}Z &= \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ !(xx) \\ \underbrace{Z \ldots (Z(Z}_{n \text{ times}}!y)) &\longrightarrow^* !(\underbrace{yy \ldots y}_{2^n \text{ times}})\end{aligned} \tag{3.5}$$

It may trigger an exponential size explosion by repeated application of the duplicating rule (!). The following term

$$\begin{aligned}&Y = \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ \S(xx) \\ &\underbrace{Y \ldots (Y(Y}_{n \text{ times}}!y)) \\ \longrightarrow^* &\underbrace{Y \ldots (Y(Y}_{n-2 \text{ times}}(\mathsf{let}\ !x = \S(yy)\ \mathsf{in}\ \S(xx)))) \nrightarrow\end{aligned} \tag{3.6}$$

respects the four criterions but cannot be used to apply (!) exponentially.

3.2 On the stratification of regions by depth

In our previous work on elementary time [16], we analyzed the impact of side effects on the depth of occurrences and remarked that arbitrary reads and writes could increase the depth of programs. In the reduction sequence

$$\begin{aligned}(\lambda x.\mathsf{set}(r, x) \parallel \S\mathsf{get}(r))!M &\longrightarrow^* \S\mathsf{get}(r) \parallel r \Leftarrow !M \\ &\longrightarrow \S!M\end{aligned} \tag{3.7}$$

the occurrence M moves from depth 1 to depth 2 during the last reduction step, because the read occurs at depth 0 while the write occurs at depth 1.

Following this analysis, we introduced *region contexts* in order to constrain the depth at which side effects occur. A region context

$$R = r_1 : \delta_1, \ldots, r_n : \delta_n$$

associates a natural number δ_i to each region r_i in a finite set of regions $\{r_1, \ldots, r_n\}$ that we write $dom(R)$. We write $R(r_i)$ for δ_i. Then, the rules of the elementary depth system were designed in such a way that $\mathsf{get}(r_i)$ and $\mathsf{set}(r_i, M)$ may only occur at depth δ_i, thus rejecting (3.7).

Moreover, we remarked that since stores are global, that is they always occur at depth 0, assigning a term to a region breaks stratification whenever $\delta_i > 0$. Indeed, in the reduction

$$\S\mathsf{set}(r, M) \longrightarrow \S\star \parallel r \Leftarrow M \tag{3.8}$$

where $R(r)$ should be 1, the occurrence M moves from depth 1 to depth 0. Therefore, we revised the definition of depth as follows.

Definition 3. *Let P be a program and R a region context where $dom(R)$ contains all the regions of P. The* revised depth $d(w)$ *of an occurrence w of P is the number of $\dagger$ labels that the path leading to the end node crosses, plus $R(r)$ if the path crosses a store label $r \Leftarrow$. The revised depth $d(P)$ of a program P is the maximum revised depth of its occurrences.*

By considering this revised definition of depth, in (3.8) the occurrence M stays at depth 1. In Figure 3 we now get $d(01000) = d(01010) = 1$, $d(10) = R(r)$ and $d(100) = d(1000) = d(10000) = d(10001) = R(r) + 1$. Other occurrences have depth 0. From now on we shall say depth for the revised definition of depth.

3.3 Inference rules

Now we introduce the inference rules of the polynomial depth system. First, we define region contexts R and variable contexts Γ as follows:

$$\begin{array}{rcl} R & = & r_1 : \delta_1, \ldots, r_n : \delta_n \\ \Gamma & = & x_1 : u_1, \ldots, x_n : u_n \end{array}$$

Regions contexts are described in the previous subsection. A variable context associates each variable with a usage $u \in \{\lambda, \S, !\}$ which constrains the variable to be bound by a λ-abstraction, a let §-binder or a let !-binder respectively. We write Γ_u if $dom(\Gamma)$ only contains variables with usage u. A depth judgement has the shape

$$R; \Gamma \vdash^{\delta} P$$

where δ is a natural number. It should entail the following:

- if $x : \lambda \in \Gamma$ then x occurs at depth δ in $\dagger^{\delta} P$,
- if $x : \dagger \in \Gamma$ then x occurs at depth $\delta + 1$ in $\dagger^{\delta} P$,
- if $r : \delta' \in R$ then $\mathsf{get}(r)$/$\mathsf{set}(r)$ occur at depth δ' in $\dagger^{\delta} P$.

The inference rules of the depth system are presented in Figure 5. We comment on the handling of usages. Variables are introduced with usage λ. The construction of !-terms updates the usage of variables to ! if they all previously had usage λ. The construction of §-terms updates the usage of variables to § for one part and ! for the other part if they all previously had usage λ. In both constructions, contexts with other usages can be weakened. As a result, λ-abstractions bind variables occurring at depth 0, let !-binders bind variables occurring at depth 1 in !-terms or §-terms, and let §-binders bind variables occurring at depth 1 in §-terms.

To control the duplication of data, the rules for binders have predicates which specify how many occurrences can be bound. λ-abstractions and let §-binders are linear by predicate $\mathsf{FO}(x, M) = 1$ and let !-binders are at least linear by predicate $\mathsf{FO}(x, M) \geq 1$.

The depth δ of the judgement is decremented when constructing $\dagger$-terms. This allows to stratify regions by depth level by requiring that $\delta = R(r)$ in the rules for $\mathsf{get}(r)$ and $\mathsf{set}(r, M)$. A store assignment $r \Leftarrow M$ is global hence its judgement has depth 0 whereas the premise has depth $R(r)$ (this reflects the revised notion of depth).

Definition 4. *(Well-formedness) A program P is* well-formed *if a judgement $R; \Gamma \vdash^{\delta} P$ can be derived for some R, Γ and δ.*

$$\frac{x : \lambda \in \Gamma}{R; \Gamma \vdash^{\delta} x} \qquad \frac{}{R; \Gamma \vdash^{\delta} \star} \qquad \frac{}{R; \Gamma \vdash^{\delta} r}$$

$$\frac{\mathsf{FO}(x, M) = 1 \quad R; \Gamma, x : \lambda \vdash^{\delta} M}{R; \Gamma \vdash^{\delta} \lambda x.M} \qquad \frac{R; \Gamma \vdash^{\delta} M \quad R; \Gamma \vdash^{\delta} N}{R; \Gamma \vdash^{\delta} MN}$$

$$\frac{\mathsf{FO}(M) \leq 1 \quad R; \Gamma_{\lambda} \vdash^{\delta+1} M}{R; \Gamma_{!}, \Delta_{\S}, \Psi_{\lambda} \vdash^{\delta} !M} \qquad \frac{\mathsf{FO}(x, N) \geq 1 \quad R; \Gamma \vdash^{\delta} M \quad R; \Gamma, x : ! \vdash^{\delta} N}{R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = M\ \mathsf{in}\ N}$$

$$\frac{R; \Gamma_{\lambda}, \Delta_{\lambda} \vdash^{\delta+1} M}{R; \Gamma_{!}, \Delta_{\S}, \Psi_{\lambda} \vdash^{\delta} \S M} \qquad \frac{\mathsf{FO}(x, N) = 1 \quad R; \Gamma \vdash^{\delta} M \quad R; \Gamma, x : \S \vdash^{\delta} N}{R; \Gamma \vdash^{\delta} \mathsf{let}\ \S x = M\ \mathsf{in}\ N}$$

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

$$\frac{r : \delta \in R \quad R; \Gamma \vdash^{\delta} M}{R; \Gamma \vdash^{0} r \Leftarrow M} \qquad \frac{i = 1, 2 \quad R; \Gamma \vdash^{\delta} P_i}{R; \Gamma \vdash^{\delta} (P_1 \parallel P_2)}$$

Figure 5. A polynomial depth system

Example 1. *The program P of Figure 3 is well-formed by composition of the two derivation trees of Figure 6. The program Z given in* (3.5) *is not well-formed.*

The depth system is strictly linear in the sense that it is not possible to bind 0 occurrences. We shall see in Section 4 that it allows for a major simplification of the proof of simulation. However, this impossibility to discard data is a notable restriction over light λ-terms. In a call-by-value setting, the sequential composition $M; N$ is usually encoded as the non well-formed term $(\lambda z.N)M$ where $z \notin \mathsf{FV}(N)$ is used to discard the terminal value of M. We show that side effects can be used to simulate the discarding of data even though the depth system is strictly linear. Assume that we dispose of a specific region gr collecting 'garbage' values at each depth level of a program. Then $M; N$ could be encoded as the well-formed program $(\lambda z.\mathsf{set}(gr, z) \parallel N)M$. Using a call-by-value semantics, we would observe the following reduction sequence

$$\begin{array}{rl} M; N \longrightarrow^{*} V; N \longrightarrow \mathsf{set}(gr, V) \parallel N \longrightarrow \star \parallel N \parallel gr \Leftarrow V & \\ \longrightarrow N \parallel gr \Leftarrow V & \end{array}$$

where $\star$ has been erased by (gc) and V has been garbage collected into gr.

Finally we derive the following lemmas on the depth system in order to get the subject reduction proposition.

Lemma 1 (Weakening and Substitution)**.**

1. *If $R; \Gamma \vdash^{\delta} P$ then $R; \Gamma, \Gamma' \vdash^{\delta} P$.*
2. *If $R; \Gamma, x : \lambda \vdash^{\delta} M$ and $R; \Gamma \vdash^{\delta} N$ then $R; \Gamma \vdash^{\delta} M[N/x]$.*
3. *If $R; \Gamma, x : \S \vdash^{\delta} M$ and $R; \Gamma \vdash^{\delta} \S N$ then $R; \Gamma \vdash^{\delta} M[N/x]$.*
4. *If $R; \Gamma, x : ! \vdash^{\delta} M$ and $R; \Gamma \vdash^{\delta} !N$ then $R; \Gamma \vdash^{\delta} M[N/x]$.*

Proposition 1 (Subject reduction)**.** *If $R; \Gamma \vdash^{\delta} P$ and $P \longrightarrow P'$ then $R; \Gamma \vdash^{\delta} P'$ and $d(P) \geq d(P')$.*

$$\dfrac{\dfrac{}{r:0;-\vdash^0 r}\quad \dfrac{\dfrac{}{r:0;x:!\vdash^0 r}\quad \dfrac{\dfrac{}{r:0;x:\lambda\vdash^1 x}\quad \dfrac{}{r:0;x:\lambda\vdash^1 x}}{r:0;x:!\vdash^0 !x}\quad \dfrac{}{r:0;x:!\vdash^0 \S x}}{r:0;x:!\vdash^0 !x\S x}}{r:0;x:!\vdash^0 \mathsf{set}(r,!x\S x)}}{r:0;-\vdash^0 \mathsf{get}(r)}$$
$$\dfrac{}{r:0;-\vdash^0 \mathsf{let}\ !x=\mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r,!x\S x)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{r:0;x:\lambda\vdash^1 x}\quad \dfrac{}{r:0;x:\lambda\vdash^1 \star}}{r:0;x:\lambda\vdash^1 x\star}}{r:0;-\vdash^1 \lambda x.x\star}}{r:0;-\vdash^0 !(\lambda x.x\star)}}{r:0;-\vdash^0 r \Leftarrow !(\lambda x.x\star)}$$

Figure 6. Derivation trees

4. Simulation by shallow-first

In this section, we first explain why we need a class of *outer-bang* reduction strategies (Subsection 4.1). Then, we prove that shallow-first simulates any outer-bang strategy and that the result applies to call-by-value (Subsection 4.2).

4.1 Towards outer-bang strategies

Reordering a reduction sequence into a shallow-first one is an iterating process where each iteration consists in commuting two consecutive reduction steps which are applied in 'deep-first' order.

First, we show that this process requires a reduction which is strictly larger than an usual call-by-value relation. Informally, assume $\dagger V$ denotes a value. The following two reduction steps in call-by-value style

$$\mathsf{set}(r,\dagger M) \xrightarrow{1} \mathsf{set}(r,\dagger V) \xrightarrow{0} \star \parallel r \Leftarrow \dagger V$$

commute into the shallow-first sequence

$$\mathsf{set}(r,\dagger M) \xrightarrow{0} \star \parallel r \Leftarrow \dagger M \xrightarrow{1} \star \parallel r \Leftarrow \dagger V$$

which is obviously not call-by-value: first, we write a non-value $\dagger M$ to the store and second we reduce *in* the store! As another example, the following two reduction steps in call-by-value style

$$(\lambda x.\lambda y.xy)\dagger M \xrightarrow{1} (\lambda x.\lambda y.xy)\dagger V \xrightarrow{0} \lambda y.(\dagger V)y$$

commute into the shallow-first sequence

$$(\lambda x.\lambda y.xy)\dagger M \xrightarrow{0} \lambda y.(\dagger M)y \xrightarrow{i} \lambda y.(\dagger V)y$$

which is not call-by-value: we need to reduce inside a λ-abstraction and this is not compatible with the usual notion of value.

Second, we show that an arbitrary relation like $\longrightarrow$ is too large to be simulated by shallow-first sequences. For instance, consider the following reduction of a well-formed program:

$$\begin{aligned} &\mathsf{let}\ !x=!\mathsf{get}(r)\ \mathsf{in}\ \S(xx) \parallel r \Leftarrow M \\ &\xrightarrow{1} \mathsf{let}\ !x=!M\ \mathsf{in}\ \S(xx) \\ &\xrightarrow{0} \S(MM) \end{aligned} \tag{4.1}$$

This sequence is deep-first; it can be reordered into a shallow-first one as follows:

$$\begin{aligned} &\mathsf{let}\ !x=!\mathsf{get}(r)\ \mathsf{in}\ \S(xx) \parallel r \Leftarrow M \\ &\xrightarrow{0} \S(\mathsf{get}(r)\mathsf{get}(r)) \parallel r \Leftarrow M \\ &\xrightarrow{1} \S(M\mathsf{get}(r)) \nrightarrow \end{aligned} \tag{4.2}$$

However, the sequence cannot be confluent with the previous one for we try to read the region two times by duplicating the redex $\mathsf{get}(r)$. It turns out that a non shallow-first strategy may require exponential time in the presence of side effects. Consider the well-formed λ-abstraction

$$F=\lambda x.\mathsf{let}\ \S x=x\ \mathsf{in}\ \S\mathsf{set}(r,x);!\mathsf{get}(r)$$

which transforms a §-term into a !-term (think of the type $\S A \multimap !A$ that would be rejected in **LLL**). Then, building on program Z given in (3.5), take

$$Z'=\lambda x.\mathsf{let}\ !x=x\ \mathsf{in}\ F\S(xx)$$

We observe an exponential explosion of the size of the following well-formed program:

$$\begin{aligned} &\underbrace{Z'Z'\dots Z'}_{n\text{ times}}!\star \\ &\longrightarrow^* \underbrace{Z'Z'\dots Z'}_{n-1\text{ times}}(F\S(\star\star)) \\ &\longrightarrow^* \underbrace{Z'Z'\dots Z'}_{n-1\text{ times}}(!(\star\star)) \parallel gr \Leftarrow \S\star \\ &\longrightarrow^* !(\underbrace{\star\star\dots\star}_{2^n\text{ times}}) \parallel \underbrace{gr \Leftarrow \S\star \parallel \dots \parallel gr \Leftarrow \S\star}_{n\text{ times}} \end{aligned}$$

where gr is a region collecting the garbage produced by the sequential composition operator of F. This previous sequence is not shallow-first since the redexes $\mathsf{set}(r,M)$ and $\mathsf{get}(r)$ occurring at depth 1 are alternatively applied with other redexes occurring at depth 0. A shallow-first strategy would produce the reduction sequence

$$\underbrace{Z'Z'\dots Z'}_{n\text{ times}}!\star \longrightarrow^* !(\star\star\underbrace{\mathsf{get}(r)\mathsf{get}(r)\dots\mathsf{get}(r)}_{n-1\text{ times}}) \parallel S$$

where S is the same garbage store as previously but we observe no size explosion.

Following these observations, our contribution is to identify an intermediate *outer-bang* reduction relation that can be simulated by shallow-first sequences. The keypoint is to prevent reductions inside !-terms like in sequence (4.1). For this, we define the *outer-bang* evaluation contexts F in Figure 7. They are not decomposable

$$\begin{aligned} F \quad ::= \quad & [\cdot] \mid \lambda x.F \mid FM \mid MF \mid \S F \\ & \mathsf{let}\ \dagger x=F\ \mathsf{in}\ M \mid \mathsf{let}\ \dagger x=M\ \mathsf{in}\ F \\ & \mathsf{set}(r,F) \mid (F \parallel M) \mid (M \parallel F) \mid r \Leftarrow F \end{aligned}$$

Figure 7. Outer-bang evaluation contexts

in a context of the shape $E[!E']$ and thus cannot be used to reduce in !-terms. In the sequel, $\longrightarrow_{\mathsf{ob}}$ denotes reduction modulo evaluation contexts F.

4.2 Simulation of outer-bang strategies

After identifying a proper outer-bang relation $\longrightarrow_{\mathsf{ob}}$, the main difficulty is to preserve the evaluation order of side effects by shallow-first reordering. For example, the following two reduction steps do not commute:

$$\begin{aligned} &F_1[\mathsf{set}(r,Q)] \parallel F_2[\mathsf{get}(r)] \\ &\xrightarrow{i} F_1[\star] \parallel F_2[\mathsf{get}(r)] \parallel r \Leftarrow Q \\ &\xrightarrow{j} F_1[\star] \parallel F_2[Q] \end{aligned} \tag{4.3}$$

We claim that this is not an issue since the depth system enforces that side effects on a given region can only occur at fixed depth, hence that $i = j$. Therefore, we should never need to 'swap' a read with a write on the same region.

We can prove the following crucial lemma.

Lemma 2 (Swapping)**.** *Let P be a well-formed program such that $P \xrightarrow{i}_{\mathsf{ob}} P_1 \xrightarrow{j}_{\mathsf{ob}} P_2$ and $i > j$. Then, there exists P' such that $P \xrightarrow{j}_{\mathsf{ob}} P' \xrightarrow{i}_{\mathsf{ob}} P_2$.*

Proof. We write M the contractum of the reduction $P \xrightarrow{i}_{\mathsf{ob}} P_1$ and N the redex of the reduction $P_1 \xrightarrow{j}_{\mathsf{ob}} P_2$. Assume they occur at addresses w_m and w_n in P_1. We distinguish three cases: (1) M and N are separated (neither $w_m \sqsubseteq w_n$ nor $w_m \sqsupseteq w_n$); (2) M contains N ($w_m \sqsubseteq w_n$); (3) N strictly contains M ($w_m \sqsupseteq w_n$ and $w_m \neq w_n$). For each of them we discuss a crucial subcase:

1. Assume M is the contractum of a (set) rule and that N is the redex of a (get) rule related to the same region. This case has been introduced in example (4.3) where M and N are separated by a parallel node. By well-formedness of P, the redexes $\mathsf{get}(r)$ and $\mathsf{set}(r, Q)$ must occur at the same depth, that is $i = j$, and we conclude that we do not need to swap the reductions.
2. If the contractum M contains the redex N, N may not exist yet in P which makes the swapping impossible. We remark that, for any well-formed program Q such that $Q \xrightarrow{d}_{\mathsf{ob}} Q'$, both the redex and the contractum occur at depth d. In particular, this is true when a contractum occurs in the store as follows:
$$Q = F[\mathsf{set}(r, T)] \xrightarrow{d}_{\mathsf{ob}} Q' = F[\star] \parallel r \Leftarrow T$$
By well-formedness of Q, there exists a region context R such that $R(r) = d$ and the redex $\mathsf{set}(r, T)$ occurs at depth d. By the revised definition of depth, the contractum T occurs at depth d in the store. As a result of this remark, M occurs at depth i and N occurs at depth j. Since $i > j$, it is clear that the contractum M cannot contain the redex N and this case is void.
3. Let N be the redex $\mathsf{let}\ \S x = \S R\ \mathsf{in}\ Q$ and let the contractum M appears in R as in the following reduction sequence
$$\begin{aligned} P &= F[\mathsf{let}\ \S x = \S R'\ \mathsf{in}\ Q] \\ &\xrightarrow{i}_{\mathsf{ob}} P_1 = F[\mathsf{let}\ \S x = \S R\ \mathsf{in}\ Q] \\ &\xrightarrow{j}_{\mathsf{ob}} P_2 = F[Q[R/x]] \end{aligned}$$
By well-formedness, x occurs exactly once in Q. This implies that applying first $P \xrightarrow{j} P'$ cannot discard the redex in R'. Hence, we can produce the following shallow-first sequence of the same length:
$$\begin{aligned} P = F[\mathsf{let}\ \S x = \S R'\ \mathsf{in}\ Q] &\xrightarrow{j}_{\mathsf{ob}} P' = F[Q[R'/x]] \\ &\xrightarrow{i}_{\mathsf{ob}} P_2 = F[Q[R/x]] \end{aligned}$$
Moreover, the reduction $P' \xrightarrow{i}_{\mathsf{ob}} P_2$ must be outer-bang for x cannot occur in a !-term in Q. □

There are two notable differences with Terui's swapping procedure. First, our procedure returns sequences of exactly the same length as the original ones while his may return longer sequences. The reason is that outer-bang contexts force redexes to be duplicated before being reduced, as in reduction (4.2), hence our swapping procedure cannot lengthen sequences more. The other difference is that his calculus is affine whereas ours is strictly linear. Therefore his procedure might shorten sequences by discarding redexes and this breaks the argument for *strong* polynomial termination. His solution is to introduce an auxiliary calculus with explicit discarding for which swapping lengthens sequences. This is at the price of introducing commutation rules which require quite a lot of extra work to obtain the simulation result. We conclude that strict linearity brings major proof simplifications while we have seen it does not cause a loss of expressivity if we use garbage collecting regions.

Using the swapping lemma, we show that any reduction sequence that uses outer-bang evaluation contexts can be simulated by a shallow-first sequence.

Proposition 2 (Simulation by shallow-first)**.** *To any reduction sequence $P_1 \xrightarrow{*}_{\mathsf{ob}} P_n$ corresponds a shallow-first reduction sequence $P_1 \xrightarrow{*}_{\mathsf{ob}} P_n$ of the same length.*

Proof. By simple application of the bubble sort algorithm: traverse the original sequence from P_1 to P_n, compare the depth of each consecutive reduction steps, swap them by Lemma 2 if they are in deep-first order. Repeat the traversal until no swap is needed. Note that we never need to swap two reduction steps of the same depth, which implies that we never need to reverse the order of dependent side effects. For example, in Figure 8, the sequence $P \xrightarrow{2}_{\mathsf{ob}} P' \xrightarrow{1}_{\mathsf{ob}} P'' \xrightarrow{0}_{\mathsf{ob}} P'''$ is reordered into $P \xrightarrow{0}_{\mathsf{ob}} C \xrightarrow{1}_{\mathsf{ob}} B \xrightarrow{2}_{\mathsf{ob}} P'''$ by 3 traversals. □

$$\begin{array}{lllllll} P & \xrightarrow{2}_{\mathsf{ob}} & P' & \xrightarrow{1}_{\mathsf{ob}} & P'' & \xrightarrow{0}_{\mathsf{ob}} & P''' \\ P & \xrightarrow{1}_{\mathsf{ob}} & A & \xrightarrow{2}_{\mathsf{ob}} & P'' & \xrightarrow{0}_{\mathsf{ob}} & P''' \\ P & \xrightarrow{1}_{\mathsf{ob}} & A & \xrightarrow{0}_{\mathsf{ob}} & B & \xrightarrow{2}_{\mathsf{ob}} & P''' \\ P & \xrightarrow{0}_{\mathsf{ob}} & C & \xrightarrow{1}_{\mathsf{ob}} & B & \xrightarrow{2}_{\mathsf{ob}} & P''' \end{array}$$

Figure 8. Reordering of $P \xrightarrow{*}_{\mathsf{ob}} P'''$ in shallow-first

As an application, we show that the simulation result applies to a call-by-value operational semantics that we define in Figure 9. We

$$\begin{array}{llll} \text{-values} & V & ::= & x \mid \star \mid r \mid \lambda x.M \mid \dagger V \\ \text{-terms} & M & ::= & V \mid MM \mid \S M \mid \mathsf{let}\ \dagger x = M\ \mathsf{in}\ M \\ & & & \mathsf{get}(r) \mid \mathsf{set}(r, M) \mid (M \parallel M) \\ \text{-stores} & S & ::= & r \Leftarrow V \mid (S \parallel S) \\ \text{-programs} & P & ::= & M \mid S \mid (P \parallel P) \\ \text{-contexts} & F_{\mathsf{v}} & ::= & [\cdot] \mid F_{\mathsf{v}} M \mid V F_{\mathsf{v}} \mid \S F_{\mathsf{v}} \\ & & & \mathsf{let}\ \dagger x = F_{\mathsf{v}}\ \mathsf{in}\ M \mid \mathsf{set}(r, F_{\mathsf{v}}) \\ & & & (F_{\mathsf{v}} \parallel P) \mid (P \parallel F_{\mathsf{v}}) \end{array}$$

-reduction rules-

$$\begin{array}{lrcl} (\beta_{\mathsf{v}}) & F_{\mathsf{v}}[(\lambda x.M)V] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M[V/x]] \\ (!_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{let}\ !x = !V\ \mathsf{in}\ M] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M[V/x]] \\ (\S_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{let}\ \S x = \S V\ \mathsf{in}\ M] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M[V/x]] \\ (\mathsf{get}_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{get}(r)] \parallel r \Leftarrow V & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[V] \\ (\mathsf{set}_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{set}(r, V)] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[\star] \parallel r \Leftarrow V \\ (\mathsf{gc}_{\mathsf{v}}) & F_{\mathsf{v}}[\star \parallel M] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M] \end{array}$$

Figure 9. CBV syntax and operational semantics

revisit the syntax of programs with a notion of value V that may be a variable, unit, a region, a λ-abstraction or a $\dagger$-value. Terms and programs are defined as previously (see Figure 2) except that $!M$ cannot be constructed unless M is a value. Store assignments are restricted to values. Evaluation contexts F_{v} are left-to-right call-by-value (obviously we do not evaluate in stores). The call-by-value

reduction relation is denoted by $\longrightarrow_{\mathsf{v}}$ and is defined modulo F_{v} and $\equiv$.

From a programming viewpoint, we shall only duplicate values. This explains why we do not want to construct $!M$ if M is not a value.

Call-by-value contexts F_{v} are outer-bang contexts since F_{v} cannot be decomposed as $E[!E']$. This allows the relation $\longrightarrow_{\mathsf{ob}}$ to contain the relation $\longrightarrow_{\mathsf{v}}$. As a result, we obtain the following corollary.

Corollary 1 (Simulation of CBV)**.** *To any reduction sequence $P_1 \longrightarrow_{\mathsf{v}}^* P_n$ corresponds a shallow-first reduction sequence $P_1 \longrightarrow_{\mathsf{ob}}^* P_n$ of the same length.*

Remark that we may obtain a non call-by-value sequence but that the semantics of the program is preserved (we compute P_n).

5. Polynomial soundness of shallow-first

In this section we prove that well-formed programs admit polynomial bounds with a shallow-first strategy. We stress that this subsection is similar to Terui's [18]; the main difficulty has been to design the polynomial depth system such that we could adopt a similar proof method.

As a first step, we define an *unfolding* transformation on programs.

Definition 5. *(Unfolding) The* unfolding *at depth i of a program P, written $\sharp^i(P)$, is defined as follows:*

$$\begin{array}{rcl}
\sharp^i(x) & = & x \\
\sharp^i(r) & = & r \\
\sharp^i(\star) & = & \star \\
\sharp^i(\lambda x.M) & = & \lambda x.\sharp^i(M) \\
\sharp^i(MN) & = & \sharp^i(M)\sharp^i(N)
\end{array}$$

$$\sharp^i(\dagger M) = \begin{cases} \dagger\sharp^{i-1}(M) & \text{if } i > 0 \\ \dagger M & \text{if } i = 0 \end{cases}$$

$$\sharp^i(\mathsf{let}\ \ \dagger x = M\ \mathsf{in}\ N) = \begin{cases} \text{if } i = 0, M = !M' \text{ and } \dagger = ! : \\ \mathsf{let}\ !x = \underbrace{MM\ldots M}_{k \text{ times}}\ \mathsf{in}\ \sharp^0(N) \\ \text{where } k = \mathsf{FO}(x, \sharp^0(N)) \\ \\ \text{otherwise:} \\ \mathsf{let}\ \dagger x = \sharp^i(M)\ \mathsf{in}\ \sharp^i(N) \end{cases}$$

$$\begin{array}{rcl}
\sharp^i(\mathsf{get}(r)) & = & \mathsf{get}(r) \\
\sharp^i(\mathsf{set}(r, M)) & = & \mathsf{set}(r, \sharp^i(M)) \\
\sharp^i(r \Leftarrow M) & = & r \Leftarrow \sharp^i(M) \\
\sharp^i(P_1 \parallel P_2) & = & \sharp^i(P_1) \parallel \sharp^i(P_2)
\end{array}$$

This unfolding procedure is intended to duplicate statically the occurrences that will be duplicated by redexes occurring at depth i. For example, in the following reductions occurring at depth 0:

$$\begin{array}{l}
P = \mathsf{let}\ !x = !M\ \mathsf{in}\ (\mathsf{let}\ !y = !x\ \mathsf{in}\ \S(yy) \parallel \mathsf{let}\ !y = !x\ \mathsf{in}\ \S(yy)) \\
\overset{0}{\longrightarrow}{}^*\ \S(MM) \parallel \S(MM)
\end{array}$$

the well-formed program P duplicates the occurrence M four times. We observe that the unfolding at depth 0 of P reflects this duplication:

$$\begin{array}{l}
\sharp^0(P) = \mathsf{let}\ !x = !M!M!M!M\ \mathsf{in} \\
\qquad (\mathsf{let}\ !y = !x!x\ \mathsf{in}\ \S(yy) \parallel \mathsf{let}\ !y = !x!x\ \mathsf{in}\ \S(yy))
\end{array}$$

Unfolded programs are not intended to be reduced. However, the size of an unfolded program can be used as a non increasing measure in the following way.

Lemma 3. *Let P be a well-formed program such that $P \overset{i}{\longrightarrow} P'$. Then $|\sharp^i(P')| \leq |\sharp^i(P)|$.*

Proof. First, we assume the occurrences labelled with '$\parallel$' and '$r \Leftarrow$' do not count in the size of a program and that '$\mathsf{set}(r)$' counts for two occurrences, such that the size strictly decreases by the rule (set). Then, it is clear that $(!)$ is the only reduction rule that can make the size of a program increase, so let

$$P = F[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] \overset{i}{\longrightarrow} P' = F[M[N/x]]$$

We have

$$\sharp^i(P) = F'[\mathsf{let}\ !x = \underbrace{!N!N\ldots!N}_{n \text{ times}}\ \mathsf{in}\ \sharp^0(M)]$$

$$\sharp^i(P') = F'[\sharp^0(M[N/x])]$$

for some context F' and $n = \mathsf{FO}(x, \sharp^0(M))$. Therefore we are left to show

$$|\sharp^0(M[N/x])| \leq |\mathsf{let}\ !x = \underbrace{!N!N\ldots!N}_{n \text{ times}}\ \mathsf{in}\ \sharp^0(M)|$$

which is clear since N must occur n times in $\sharp^0(M[N/x])$. □

We observe in the following lemma that the size of an unfolded program bounds quadratically the size of the original program.

Lemma 4. *If P is well-formed, then for any depth $i \leq d(P)$:*

1. $\mathsf{FO}(\sharp^i(P)) \leq |P|$,
2. $|\sharp^i(P)| \leq |P| \cdot (|P| - 1)$,

Proof. By induction on P and i. □

We can then bound the size of a program after reduction.

Lemma 5 (Squaring)**.** *Let P be a well-formed program such that $P \overset{i}{\longrightarrow}{}^* P'$. Then:*

1. $|P'| \leq |P| \cdot (|P| - 1)$
2. *the length of the sequence is bounded by $|P|$*

Proof.

1. By Lemma 3 it is clear that $|\sharp^i(P')| \leq |\sharp^i(P)|$. Then by Lemma 4-2 we obtain $|\sharp^i(P')| \leq |P| \cdot (|P| - 1)$. Finally it is clear that $|P'| \leq |\sharp^i(P')|$ thus $|P'| \leq |P| \cdot (|P| - 1)$.
2. It suffices to remark $|P'|_i < |P|_i \leq |P|$. □

Finally we obtain the following theorem for a shallow-first strategy using any evaluation context.

Theorem 1 (Polynomial bounds)**.** *Let P be a well-formed program such that $d(P) = d$ and $P \longrightarrow^* P'$ is shallow-first. Then:*

1. $|P'| \leq |P|^{2^d}$
2. *the length of the reduction sequence is bounded by $|P|^{2^d}$*

Proof. The reduction $P \longrightarrow^* P'$ can be decomposed as $P = P_0 \overset{0}{\longrightarrow}{}^* P_1 \overset{1}{\longrightarrow}{}^* \ldots \overset{d-1}{\longrightarrow}{}^* P_d \overset{d}{\longrightarrow}{}^* P_{d+1} = P'$. To prove (1), we observe that by iterating Lemma 5-1 we obtain $|P_d| \leq |P_0|^{2^d}$. Moreover it is clear that $|P_{d+1}| \leq |P_d|$. Hence $|P'| \leq |P|^{2^d}$. To prove (2), we first prove by induction on d that $|P_0| + |P_1| + \ldots + |P_d| \leq |P_0|^{2^d}$. By Lemma 5-2, it is clear that the length of the

reduction $P \longrightarrow^* P'$ is bounded by $|P_0| + |P_1| + \ldots + |P_d|$, which is in turn bounded by $|P_0|^{2^d}$. □

It is worth noticing that the first bound takes the size of all the threads into account and that the second bound is valid for any thread interleaving.

Corollary 2 (Call-by-value is polynomial)**.** *The call-by-value evaluation of a well-formed program P of size n and depth d can be computed in time $O(n^{2^d})$.*

Proof. Let $P \longrightarrow_{\mathsf{v}}^* P'$ be the call-by-value reduction sequence of the well-formed program P. By Corollary 1 we can reorder the sequence into a shallow-first sequence $P \longrightarrow_{\mathsf{ob}}^* P'$ of the same length. By Theorem 1 we know that its length is bounded by $|P|^{2^d}$ and that $|P'| \leq |P|^{2^d}$. □

6. A polynomial type system

The depth system entails termination in polynomial time but does *not* guarantee that programs 'do not go wrong'. In particular, the well-formed program in (3.6) get stuck on a non-value. In this section, we propose a solution to this problem by introducing a polynomial type system as a simple decoration of the polynomial depth system with linear types. Then, we derive a progress proposition which guarantees that well-typed programs cannot deadlock (except when trying to read an empty region).

We define the syntax of types and contexts in Figure 10. Types

-type variables	$t, t', \ldots$		
-types	α	$::=$	$\mathbf{B} \mid A$
-res. types	A	$::=$	$t \mid \mathbf{1} \mid A \multimap \alpha \mid \dagger A \mid \forall t.A \mid \mathsf{Reg}_r A$
-var. contexts	Γ	$::=$	$x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)$
-reg. contexts	R	$::=$	$r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)$

Figure 10. Syntax of types, effects and contexts

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 result (such as a store or several terms in parallel) while types of entities that may return a result are denoted with A. Among the types A, we distinguish type variables $t, t', \ldots$, a terminal type $\mathbf{1}$, a linear functional type $A \multimap \alpha$, the type $!A$ of terms of type A that may be duplicated, the type $\S A$ of terms of type A that may have been duplicated, the type $\forall t.A$ of polymorphic terms and the type $\mathsf{Reg}_r A$ of regions r containing terms of type A. Hereby types may depend on regions.

In contexts, usages play the same role as in the depth system. Writing $x : (u, A)$ means that the variable x ranges on terms of type A and can be bound according to u. Writing $r : (\delta, A)$ means that the region r contain terms of type A and that $\mathsf{get}(r)$ and $\mathsf{set}(r, M)$ may only occur at depth δ. The typing system will additionally guarantee that whenever we use a type $\mathsf{Reg}_r A$ the region context contains a hypothesis $r : (\delta, A)$.

Because types depend on regions, we have to be careful in stating in Figure 11 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 constants 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 $B = A_i$.

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

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

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

Figure 11. Types and contexts

Example 2. *One may verify that the judgment $r : (\delta, \mathbf{1} \multimap \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$ can be derived while judgements $r : (\delta, \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$ and $r : (\delta, \mathsf{Reg}_r \mathbf{1}) \vdash \mathbf{1}$ cannot.*

We notice the following substitution property on types.

Proposition 3. *If $R \vdash \forall t.A$ and $R \vdash B$ then $R \vdash A[B/t]$.*

A typing judgement takes the form: $R; \Gamma \vdash^\delta P : \alpha$. It attributes a type α to the program P occurring at depth δ, according to region context R and variable context Γ. Figure 12 introduces the polynomial type system. We comment on some of the rules. A λ-abstraction may only take a term of result-type as argument, *i.e.* two threads in parallel are not considered an argument. The typing of $\dagger$-terms is limited to result-types for we may not duplicate several threads in parallel. There exists two rules for typing parallel programs. The one on the left indicates that a program P_2 in parallel with a store or a thread producing a terminal value should have the type of P_2 since we might be interested in its result (note that we omit the symmetric rule for the program $(P_2 \parallel P_1)$). The one on the right indicates that two programs in parallel cannot reduce to a single result.

Example 3. *The program of Figure 3 is well-typed according to the following derivable judgement:*

$$R; - \vdash^\delta \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, (!x)(\S x)) \parallel r \Leftarrow\ !(\lambda x.x\star) : \mathbf{1}$$

where $R = r : (\delta, \forall t.!((\mathbf{1} \multimap t) \multimap t))$. Whereas the program in (3.6) *is not.*

Remark 1. *We can easily see that a well-typed program is also well-formed.*

The polynomial type system enjoys the subject reduction property for the largest relation $\longrightarrow \supseteq \longrightarrow_{\mathsf{ob}} \supseteq \longrightarrow_{\mathsf{v}}$.

Lemma 6 (Substitution)**.**

1. *If $R; \Gamma, x : (\lambda, A) \vdash^\delta M : B$ and $R; \Gamma \vdash^\delta N : A$ then $R; \Gamma \vdash^\delta M[N/x] : B$.*
2. *If $R; \Gamma, x : (\S, A) \vdash^\delta M : B$ and $R; \Gamma \vdash^\delta \S N : \S A$ then $R; \Gamma \vdash^\delta M[N/x] : B$.*
3. *If $R; \Gamma, x : (!, A) \vdash^\delta M : B$ and $R; \Gamma \vdash^\delta !N : !A$ then $R; \Gamma \vdash^\delta M[N/x] : B$.*

Proposition 4 (Subject Reduction)**.** *If $R; \Gamma \vdash^\delta P : \alpha$ and $P \longrightarrow P'$ then $R; \Gamma \vdash^\delta P' : \alpha$.*

$$\frac{R \vdash \Gamma \quad x : (\lambda, A) \in \Gamma}{R; \Gamma \vdash^\delta x : A} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash^\delta \star : \mathbf{1}} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash^\delta r : \mathsf{Reg}_r A} \qquad \frac{\mathsf{FO}(x, M) = 1 \quad R; \Gamma, x : (\lambda, A) \vdash^\delta M : \alpha}{R : \Gamma \vdash^\delta \lambda x.M : A \multimap \alpha}$$

$$\frac{R; \Gamma \vdash^\delta M : A \multimap \alpha \quad R; \Gamma \vdash^\delta N : A}{R; \Gamma \vdash^\delta MN : \alpha} \qquad \frac{\mathsf{FO}(M) \leq 1 \quad R; \Gamma_\lambda \vdash^{\delta+1} M : A}{R; \Gamma_!, \Delta_\S, \Psi_\lambda \vdash^\delta !M : !A} \qquad \frac{R; \Gamma \vdash^\delta M : !A \quad \mathsf{FO}(x, N) \geq 1 \quad R; \Gamma, x : (!, A) \vdash^\delta N : \alpha}{R; \Gamma \vdash^\delta \mathsf{let}\ !x = M\ \mathsf{in}\ N : \alpha} \qquad \frac{R; \Gamma_\lambda, \Delta_\lambda \vdash^{\delta+1} M : A}{R; \Gamma_\S, \Delta_!, \Psi_\lambda \vdash^\delta \S M : \S A}$$

$$\frac{R; \Gamma \vdash^\delta M : \S A \quad \mathsf{FO}(x, N) = 1 \quad R; \Gamma, x : (\S, A) \vdash^\delta N : \alpha}{R; \Gamma \vdash^\delta \mathsf{let}\ \S x = M\ \mathsf{in}\ N : \alpha} \qquad \frac{t \notin (R; \Gamma) \quad R; \Gamma \vdash^\delta M : A}{R; \Gamma \vdash^\delta M : \forall t.A} \qquad \frac{R; \Gamma \vdash^\delta M : \forall t.A \quad R \vdash B}{R; \Gamma \vdash^\delta M : A[B/t]} \qquad \frac{R \vdash \Gamma \quad r : (\delta, A) \in R}{R; \Gamma \vdash^\delta \mathsf{get}(r) : A}$$

$$\frac{r : (\delta, A) \quad R; \Gamma \vdash^\delta M : A}{R; \Gamma \vdash^\delta \mathsf{set}(r, M) : \mathbf{1}} \qquad \frac{r : (\delta, A) \quad R; \Gamma \vdash^\delta M : A}{R; \Gamma \vdash^0 r \Leftarrow M : \mathbf{B}} \qquad \frac{R; \Gamma \vdash^\delta P_1 : \mathbf{1} \text{ or } P_1 = S \quad R; \Gamma \vdash^\delta P_2 : \alpha}{R; \Gamma \vdash^\delta (P_1 \parallel P_2) : \alpha} \qquad \frac{R; \Gamma \vdash^\delta P_i : \alpha_i}{R; \Gamma \vdash^\delta (P_1 \parallel P_2) : \mathbf{B}}$$

Figure 12. A polynomial type system

Finally, we establish a progress proposition which shows that any well-typed call-by-value program (*i.e.* defined from Figure 9) reduces to several threads in parallel which are values or deadlocking reads.

Proposition 5 (Progress)**.** *Suppose P is a closed typable call-by-value program which cannot reduce. Then P is structurally equivalent to a program*

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

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

7. Expressivity

We now illustrate the expressivity of the polynomial type system. First we show that our system is complete in the extensional sense: every polynomial time function can be represented (Subsection 7.1). Then we introduce a language with memory locations representing higher-order references for which the type system can be easily adapted (Subsection 7.2). Building on this language, we give an example of polynomial programming (Subsection 7.3).

As a first step, we define some Church-like encodings in Figure 13 where we abbreviate $\lambda x.\mathsf{let}\ \dagger x = x\ \mathsf{in}\ M$ by $\lambda^\dagger x.M$. We have natural numbers of type Nat, binary natural number of type BNat and lists of type $\mathsf{List}\ A$ that contain values of type A.

7.1 Polynomial completeness

The representation of polynomial functions relies on the representation of binary words. The precise notion of representation is spelled out in the following definitions.

Definition 6. *(Binary word representation) Let $- \vdash^\delta M : \S^p\mathsf{BNat}$ for some $\delta, p \in \mathbb{N}$. We say M* represents *$w \in \{0,1\}^*$, written $M \Vdash w$, if $M \longrightarrow^* \S^p\overline{w}$.*

Definition 7. *(Function representation) Let $- \vdash^\delta F : \mathsf{BNat} \multimap \S^d\mathsf{BNat}$ where $\delta, d \in \mathbb{N}$ and $f : \{0,1\}^* \to \{0,1\}^*$. We say F* represents *f, written $F \Vdash f$, if for any M and $w \in \{0,1\}^*$ such that $- \vdash^\delta M : \mathsf{BNat}$ and $M \Vdash w$, $FM \Vdash f(w)$.*

The following theorem is a restatement of Girard [12] and Asperti [4].

$$\begin{array}{rcl}
\mathsf{Nat} & = & \forall t.!(t \multimap t) \multimap \S(t \multimap t) \\
\overline{n} & : & \mathsf{Nat} \\
\overline{n} & = & \lambda^! f.\S(\lambda x.\underbrace{f(\ldots(fx))}_{n \text{ times}}) \\
 & & \\
\mathsf{add} & : & \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat} \\
\mathsf{add} & = & \lambda m.\lambda n.\lambda^! f.\mathsf{let}\ \S y = m!f\ \mathsf{in} \\
 & & \mathsf{let}\ \S z = n!f\ \mathsf{in}\ \S(\lambda x.y(zx)) \\
 & & \\
\mathsf{BNat} & = & \forall t.!(t \multimap t) \multimap !(t \multimap t) \multimap \S(t \multimap t) \\
\text{for } w = i_0 \ldots i_n \in \{0,1\}^* & & \\
\overline{w} & : & \mathsf{BNat} \\
\overline{w} & = & \lambda^! x_0.\lambda x_1^!.\S(\lambda z.x_{i_0}(\ldots(x_{i_n} z))) \\
 & & \\
\mathsf{List}\ A & = & \forall t.!(A \multimap t \multimap t) \multimap \S(t \multimap t) \\
[u_1, \ldots, u_n] & : & \mathsf{List}\ A \\
[u_1, \ldots, u_n] & = & \lambda f^!.\S(\lambda x.fu_1(fu_2 \ldots (fu_n x))) \\
 & & \\
\mathsf{list_it} & : & \forall u.\forall t.!(u \multimap t \multimap t) \multimap \mathsf{List}\ u \multimap \S t \multimap \S t \\
\mathsf{list_it} & = & \lambda f.\lambda l.\lambda^\S x.\mathsf{let}\ \S y = lf\ \mathsf{in}\ \S(yx)
\end{array}$$

Figure 13. Church encodings

Theorem 2 (Polynomial completeness)**.**
Every function $f : \{0,1\}^* \to \{0,1\}^*$ which can be computed by a Turing machine in time bounded by a polynomial of degree d can be represented by a term of type $\mathsf{BNat} \multimap \S^d\mathsf{BNat}$.

7.2 A language with higher-order references

Next, we give an application of the language with abstract regions by presenting a connection with a language with dynamic memory locations representing higher-order references.

The differences with the region-based system are presented in Figure 14. We introduce terms of the form $\nu x.M$ to generate a fresh memory location x whose scope is M. Contexts are call-by-value and allow evaluation under ν binders. The structural rule (ν) is for scope extrusion. Region constants have been removed from the syntax of terms hence reduction rules (get_ν) and (set_ν) relate to memory locations. The operational semantics of references is adopted: when assigning a value to a memory location, the previous value is *overwritten*, and when reading a memory location, the

$$
\begin{array}{rcl}
M & ::= & \ldots \mid \nu x.M \\
F_\nu & ::= & F_{\mathsf{v}} \mid \nu x.F_\nu
\end{array}
$$

$$
(\nu) \qquad F_\nu[\nu x.M] \equiv \nu x.F_\nu[M] \quad \text{if } x \notin \mathsf{FV}(F_\nu)
$$

$$
\begin{array}{lrcl}
(\mathsf{get}_\nu) & F_\nu[\mathsf{get}(x)] \parallel x \Leftarrow V & \longrightarrow_\nu & F_\nu[V] \parallel x \Leftarrow V \\
(\mathsf{set}_\nu) & F_\nu[\mathsf{set}(x,V)] \parallel x \Leftarrow V' & \longrightarrow_\nu & F_\nu[\star] \parallel x \Leftarrow V
\end{array}
$$

$$
\frac{R;\Gamma, x : (u, \mathsf{Reg}_r!A) \vdash^\delta M : B}{R;\Gamma \vdash^\delta \nu x.M : B}
\qquad
\frac{R(r) = (\delta, !A) \quad R;\Gamma \vdash^\delta x : \mathsf{Reg}_r!A}{R;\Gamma \vdash^\delta \mathsf{get}(x) : !A}
$$

$$
\frac{R(r) = (\delta, !A) \quad R;\Gamma \vdash^\delta x : \mathsf{Reg}_r!A \quad R;\Gamma \vdash^\delta M : !A}{R;\Gamma \vdash^\delta \mathsf{set}(x,M) : \mathbf{1}}
\qquad
\frac{R(r) = (\delta, !A) \quad R;\Gamma \vdash^\delta x : \mathsf{Reg}_r!A \quad R;\Gamma \vdash^\delta V : !A}{R;\Gamma \vdash^0 x \Leftarrow V : \mathbf{B}}
$$

Figure 14. A call-by-value system with references

value is *copied* from the store. We see in the typing rules that region constants still appear in region types and that a memory location must be a free variable that relates to an abstract region r by having the type $\mathsf{Reg}_r A$.

There is a simple translation from the language with memory locations to the language with regions. It consists in replacing the (free or bound) variables with a region type of the shape $\mathsf{Reg}_r A$ by the constant r. We then observe that read access and assignments to references are mapped to several reduction steps in the system with regions. It requires the following observation: in the typing rules, memory locations only relate to regions with duplicable content of type $!A$. This allows us to simulate the *copy from memory* mechanism of references by decomposing it into a *consume* and *duplicate* mechanism in the language with regions. More precisely: an occurrence of $\mathsf{get}(x)$ where x relates to region r is translated into

$$\mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !y) \parallel !y$$

such that

$$
\begin{array}{l}
\quad F_{\mathsf{v}}[\mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !y) \parallel !y] \parallel r \Leftarrow !V \\
\longrightarrow_{\mathsf{v}}^+ F[!V] \parallel r \Leftarrow !V
\end{array}
$$

simulates the reduction (get_ν). Also, it is easy to see that a reduction step (set_ν) can be simulated by exactly one reduction step $(\mathsf{set}_{\mathsf{v}})$. Since typing is preserved by translation, we conclude that any time complexity bound can be lifted to the language with references.

Note that this also works if we adopt the operational semantics of communication channels; in that case, memory locations can also relate to regions containing non-duplicable content since reading a channel means *consuming* the value.

7.3 Polynomial programming

Using higher-order references, we show that it is possible to program the iteration of operations producing a side effect on an inductive data structure, possibly in parallel.

Here is the function update taking as argument a memory location x related to region r and incrementing the numeral stored at that location:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}); - \vdash^2 \mathsf{update} : !\mathsf{Reg}_r!\mathsf{Nat} \multimap \S\mathbf{1} \multimap \S\mathbf{1} \\
\mathsf{update} = \lambda^! x.\lambda^\S z.\S(\mathsf{set}(x, \mathsf{let}\ !y = \mathsf{get}(x)\ \mathsf{in}\ !(\mathsf{add}\ \overline{2}\ y)) \parallel z)
\end{array}
$$

The second argument z is to be garbage collected. Then we define the program run that iterates the function update over a list $[!x, !y, !z]$ of 3 memory locations:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}) \vdash^1 \mathsf{run} : \S\S\mathbf{1} \\
\mathsf{run} = \mathsf{list\_it}\ !\mathsf{update}\ [!x, !y, !z]\ \S\S\star
\end{array}
$$

All addresses have type $!\mathsf{Reg}_r!\mathsf{Nat}$ and thus relate to the same region r. Finally, the program run in parallel with some store assignments reduces as expected:

$$
\begin{array}{l}
\quad \mathsf{run} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p} \\
\longrightarrow_\nu^* \S\S\star \parallel x \Leftarrow !\overline{2+m} \parallel y \Leftarrow !\overline{2+n} \parallel z \Leftarrow !\overline{2+p}
\end{array}
$$

Note that due to the Church-style encoding of numbers and lists, we assume that the relation $\longrightarrow_\nu$ may reduce under binders when required.

Building on this example, suppose we want to write a program of three threads where each thread concurrently increments the numerals pointed by the memory locations of the list. Here is the function gen_threads taking a functional f and a value x as arguments and generating three threads where x is applied to f:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}) \vdash^0 \mathsf{gen\_threads} : \forall t.\forall t'.!(t \multimap t') \multimap !t \multimap \mathbf{B} \\
\mathsf{gen\_threads} = \lambda^! f.\lambda^! x.\S(fx) \parallel \S(fx) \parallel \S(fx)
\end{array}
$$

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

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}) \vdash^1 \mathsf{F} : \mathsf{List}\ !\mathsf{Reg}_r!\mathsf{Nat} \multimap \S\S\mathbf{1} \\
\mathsf{F} = \lambda l.\mathsf{list\_it}\ !\mathsf{update}\ l\ \S\S\star
\end{array}
$$

Finally the concurrent iteration is defined in run_threads:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}) \vdash^0 \mathsf{run\_threads} : \mathbf{B} \\
\mathsf{run\_threads} = \mathsf{gen\_threads}\ !\mathsf{F}\ ![!x, !y, !z]
\end{array}
$$

The program is well-typed for side effects occurring at depth 3 and it reduces as follows:

$$
\begin{array}{l}
\quad \mathsf{run\_threads} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p} \\
\longrightarrow_\nu^* \S\S\S\star \parallel x \Leftarrow !\overline{6+m} \parallel y \Leftarrow !\overline{6+n} \parallel z \Leftarrow !\overline{6+p}
\end{array}
$$

Note that different thread interleavings are possible but in this particular case they are confluent.

8. Conclusion and Related work

We have proposed a type system for a higher-order functional language with multithreading and side effects that guarantees termination in polynomial time, covering any scheduling of threads and taking account of thread generation. To the best of our knowledge, there appears to be no other characterization of polynomial time in such a language. The polynomial soundness of the call-by-value strategy relies on the simulation of call-by-value by a shallow-first strategy which is proved to be polynomial. The proof is a significant adaptation of Terui's methodology [18]: it is greatly simplified by a strict linearity condition and based on a clever analysis of the evaluation order of side effects which is shown to be preserved.

Related work The framework of light logics has been previously applied to a higher-order π-calculus [14] and a functional language with pattern-matching and recursive definitions [6]. The notion of *stratified region*[1] has been proposed [1, 7] to ensure the termination of a higher-order multithreaded language with side effects . In the setting of *synchronous* computing, static analyses have been developed to bound resource consumption in a synchronous π-calculus [2] and a multithreaded first-order language [3]. Recently, the framework of *complexity information flow* have been applied to characterize polynomial multithreaded imperative programs [17].

[1] Here we speak of stratification by means of a type-and-effect discipline, this is not to be confused with the notion of stratification *by depth level* that is used in the present paper.

Acknowledgments The author wishes to thank Roberto Amadio for his precious help on the elaboration of this work and Patrick Baillot for his careful reading of the paper.

References

[1] R. M. Amadio. On stratified regions. In Z. Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2009. ISBN 978-3-642-10671-2. 2, 8

[2] R. M. Amadio and F. Dabrowski. Feasible reactivity in a synchronous pi-calculus. In M. Leuschel and A. Podelski, editors, *PPDP*, pages 221–230. ACM, 2007. ISBN 978-1-59593-769-8. 8

[3] R. M. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. *Theoretical Computer Science*, 358(2-3):229–254, 2006. 8

[4] A. Asperti. Light affine logic. In *LICS*, pages 300–308. IEEE Computer Society, 1998. ISBN 0-8186-8506-9. 1, 7.1

[5] P. Baillot and V. Mogbil. Soft lambda-calculus: A language for polynomial time computation. In I. Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004. ISBN 3-540-21298-1. 1

[6] P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In A. D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2010. ISBN 978-3-642-11956-9. 8

[7] G. Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6): 716–736, 2010. 8

[8] P. Coppola and S. Martini. Optimizing optimal reduction: A type inference algorithm for elementary affine logic. *ACM Transaction on Computational Logic*, 7:219–260, April 2006. ISSN 1529-3785. 1

[9] 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. 1

[10] V. Danos and J.-B. Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123 – 137, 2003. ISSN 0890-5401. 1

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

[12] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998. 1, 7.1

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

[14] U. D. Lago, S. Martini, and D. Sangiorgi. Light logics and higher-order processes. In S. B. Fröschle and F. D. Valencia, editors, *EXPRESS*, volume 41 of *EPTCS*, pages 46–60, 2010. 8

[15] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM, 1988. ISBN 0-89791-252-7. 1

[16] A. Madet and R. M. Amadio. An elementary affine λ-calculus with multithreading and side effects. In C.-H. L. Ong, editor, *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011. ISBN 978-3-642-21690-9. 1, 1, 3.2

[17] J.-Y. Marion and R. Péchoux. Complexity information flow in a multi-threaded imperative language. *CoRR*, abs/1203.6878, 2012. 8

[18] K. Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3-4):253–280, 2007. 1, 1, 3.1, 5, 8