Strong Normalisation in λ-calculi with References

Romain Demangeon[1], Daniel Hirschkoff[1], and Davide Sangiorgi[2]

[1] ENS Lyon, Université de Lyon, CNRS, INRIA, France
[2] INRIA/Università di Bologna, Italy

Abstract. We present a method for ensuring termination of lambda-calculi with references. This method makes it possible to combine measure-based techniques for termination of imperative languages with traditional approaches to termination in purely functional languages, such as logical relations. More precisely, the method lifts any termination proof for the purely functional simply-typed lambda-calculus to a termination proof for the lambda-calculus with references. The method can be made parametric on the termination technique employed for the functional core.

1 Motivations

This paper studies strong normalisation in $\lambda_{\mathtt{ref}}$, a call-by-value λ-calculus with (higher-order) references. It is well-known that, even in the simply-typed calculus, the problem is difficult, because references allow one to program loops "via the memory". We refer to Boudol's [3] for a discussion on existing works on this question.

Boudol [3] has proposed a type and effect system for a calculus whose core is very similar to $\lambda_{\mathtt{ref}}$; the system guarantees termination by means of the realisability technique. That work is revisited and generalised in [1], where the closely related technique of reducibility candidates is exploited to establish soundness of the type and effect system. In both these works, the type and effect system relies on a stratification of memory into regions; the stratification is used to control interactions between the functional and the imperative constructs, in order to prevent "loops via the memory". The stratification plays also a key role in the structure of the soundness proof, to support the induction argument. Boudol's approach has also been investigated by Tranquilli [9], who proposes an analysis of the stratification imposed by the type and effect system, by means of a monadic translation. The target of this translation, in the general case, is a lambda-calculus with recursive types. Tranquilli however shows that when applying the translation to well-typed source terms, one can avoid the use of recursive types. By combining this observation with a simulation result, the author concludes that well-typed terms terminate.

In this paper, we propose a different proof strategy for strong normalisation in $\lambda_{\mathtt{ref}}$. Our approach is adapted from [7], where we introduced a type system for termination of mobile processes. The crux in defining types in that work is to

distinguish between functional and imperative channels, and to exploit a stratification of imperative channel names. Soundness of the type system is established by defining a projection of an *impure calculus*, that is, a calculus featuring imperative and functional features, into a purely functional core calculus (in the context of the π-calculus, the functional subcalculus is given, intuitively, by the image of the encoding of the λ-calculus in the π-calculus). The proof then relies on termination of the functional core, which is treated like a "black box" in the proof: since our projection function preserves divergences, and the target calculus is terminating, we can reason by contradiction to show that the source of the translation only consists of terminating terms.

In the present paper we show that we can transport the strategy from [7] onto $\lambda_{\mathtt{ref}}$. In contrast with π-calculus, $\lambda_{\mathtt{ref}}$ is purely sequential and higher-order (it involves substitutions of variables with terms); both these features have a substantial impact on the details of the technique. In this sense, another goal of the paper is to show that the technique in [7] is not specific to a concurrent scenario, and can be used on different kinds of impure languages. The "black box" property for the purely functional subcalculus in [7] remains: the technique for $\lambda_{\mathtt{ref}}$ is essentially parametric on the method employed for ensuring termination of the pure λ-calculus (realisability, reducibility candidates or other methods). The present paper is devoted to the presentation of our technique in a rather simple setting, where the core functional language is the simply typed λ-calculus.

With respect to [7], several modifications have to be made in order to handle $\lambda_{\mathtt{ref}}$. Many of them are related to the definition of the projection function, which in the present work maps λ-terms with references to purely functional terms. In the π-calculus the projection acts on prefixed terms, simply by replacing some of them with the inactive process $\mathbf{0}$; this crucially relies on the operators of parallel composition and $\mathbf{0}$ of the process calculus. In the λ-calculus the situation is more intricate. Consider for instance a $\lambda_{\mathtt{ref}}$ term of the form $T = (\lambda z.\star)\ (\mathtt{ref}\ M)$, where $\star$ is the unique element of type $\mathbb{1}$ (the unit type), and $\mathtt{ref}\ M$ denotes the allocation of a reference holding the value of M (the – slightly more involved – syntax and operational semantics of $\lambda_{\mathtt{ref}}$ will be introduced formally below). The idea is to project T into some purely functional term T', in such a way that: (i) if T is typable in our type and effect system, then T' is typable according to simple types; (ii) the projection function is defined compositionally on the structure of terms, and preserves divergences. In a call-by-value strategy, if the evaluation of M terminates, the evaluation of T yields $\star$. In order to preserve divergences, because of a potential divergence in M, we cannot define T' by simply erasing the subterm $\mathtt{ref}\ M$. Instead, we set $T' \stackrel{\text{def}}{=} (\lambda x_1.\lambda x_2.\, x_1)\ \star\ M'$, where M' is the purely functional term obtained by applying recursively the projection to M. This way, T' diverges if M' does so, and eventually returns $\star$, in case M' converges. This shows how the projection acts at an operational level. In the proof, we also take care of condition (i) above, by defining the translation both on terms and on types, in such a way as to preserve typability.

Building on the projection function, we derive soundness of the type and effect system by contradiction: suppose a well-typed $\lambda_{\mathtt{ref}}$ term T diverges, then

its projection T' is diverging too, which contradicts the fact that T' belongs to a terminating calculus. Termination of T' is obtained by an external argument, namely the strong normalisation proof for the functional subcalculus (here, the simply typed call by value λ-calculus).

Other technical differences with respect to the technique in the setting of the π-calculus [7] are discussed later in the paper.

Comparison with [3, 1, 9]. As we hinted above, the question we address in this paper has been studied in a very similar setting in other works. In contrast with the works by Boudol and Amadio, where soundness of the type system is obtained by a 'semantic' approach (be it realisability or reducibility candidates), which is applied to the whole (impure) calculus, we somehow factor out the imperative part of the calculus, which allows us to lift a termination proof of $\lambda_{\mathbf{ST}}$ to a termination proof of $\lambda_{\mathtt{ref}}$.

Tranquilli [9] proceeds similarly, in two steps: a translation into a purely functional calculus, followed by a termination argument about the latter. However, technically, our approach and his differ considerably, in particular because we project into a subcalculus, using a translation function which seems unrelated to Tranquilli's.

Outline. We introduce $\lambda_{\mathtt{ref}}$ and its type and effect system in Section 2. Section 3 is devoted to the soundness proof, where we present in particular the projection function on $\lambda_{\mathtt{ref}}$. In Section 4, we discuss how the proof can be extended to calculi richer than simple types.

2 $\lambda_{\mathtt{ref}}$: a λ-calculus with References

2.1 Syntax and semantics for $\lambda_{\mathtt{ref}}$

We now define the calculi we manipulate in this work. The standard, simply-typed, λ-calculus with the constant $\star$ and the base type $\mathbb{1}$ is called $\lambda_{\mathbf{ST}}$ in the following. The reduction relation in $\lambda_{\mathbf{ST}}$ is full β-reduction, and is denoted using $\twoheadrightarrow$.

$\lambda_{\mathtt{ref}}$ is a call-by-value λ-calculus extended with imperative operations (read, write and update) acting on a store (sometimes called a *memory* in the following). The store is stratified into regions, which are referred to using natural numbers, i.e., we suppose that the store is divided into a finite number of regions, and that there exists an enumeration of these regions. Constructs of the language involving imperative operations are annotated by a region — thus, by a natural number. For instance, $\mathtt{deref}_n(M)$ is the operator that reads the value stored at the address which is returned by the evaluation of M; n denotes the fact that this address belongs to the region n of the memory.

To define terms of $\lambda_{\mathtt{ref}}$, we rely on a set of *addresses*, which are distinct from the variables used in the syntax of the standard λ-calculus. Addresses are written $u_{(n,T)}$: they are explicitly associated both to a region n and to a type T (types are described below). These annotations are not mandatory in order to

$$
\begin{array}{rcl}
M & ::= & (M\ M) \mid x \mid \lambda x.\, M \mid \star \\
& & \mid \mathtt{ref}_n\ M \mid \mathtt{deref}_n(M) \mid M \mathtt{:=}_n M \mid u_{(n,T)} \\
T & ::= & \mathbb{1} \mid T\ \mathtt{ref}_n \mid T \to^n T \\
V & ::= & \lambda x.\, M \mid x \mid u_{(n,T)} \mid \star \\
R & ::= & (\lambda x.\, M)\ V \\
& & \mid \mathtt{deref}_n(u_{(n,T)}) \mid \mathtt{ref}_n\ V \mid u_{(n,T)} \mathtt{:=}_n V \\
\mathbf{E} & ::= & [\,] \mid V\ \mathbf{E} \mid \mathbf{E}\ M \\
& & \mid \mathtt{deref}_n(\mathbf{E}) \mid \mathtt{ref}_n\ \mathbf{E} \mid \mathbf{E} \mathtt{:=}_n M \mid V \mathtt{:=}_n \mathbf{E}
\end{array}
$$

Fig. 1. Syntax for terms, types, values, redexes and evaluation contexts

obtain the results we state in this paper, but they improve the readability of our proofs. Note in passing that values of different types can be stored in the same region. We suppose that there exists an infinite number of addresses for a given pair consisting of a type and a region.

Stores, ranged over using δ, are formally defined as partial mappings from addresses to values. The (finite) support of δ is written $\mathrm{supp}(\delta)$, $\emptyset$ is the empty store ($\mathrm{supp}(\emptyset) = \emptyset$), and $\delta\langle u_{(n,T)} \rightsquigarrow V\rangle$ denotes the store δ' defined by $\delta'(u_{(n,T)}) = V$ and $\delta'(v) = \delta(v)$ for every $v \in \mathrm{supp}(\delta)$ such that $v \neq u_{(n,T)}$.

Figure 1 presents the grammar definitions for (respectively) terms, types, values, redexes and evaluation contexts.

The standard λ-calculus syntax is extended with the unit value ($\star$), addresses and three imperative operators. $\mathtt{ref}_n\ M$ stands for the creation of a new cell in the store, at region n, and containing the result of the evaluation of M; $\mathtt{deref}_n(M)$ yields the value that is stored at the address given by the evaluation of M (in region n); finally, $M \mathtt{:=}_n N$ updates the value stored at the address given by the evaluation of M with the value of N.

Types extend the simple types of $\lambda_{\mathbf{ST}}$ with unit ($\mathbb{1}$) and a reference type: $T\ \mathtt{ref}_n$ is the type of an address in region n containing values of type T. To record the latent effect of a function, arrow types are annotated with regions: intuitively, $T_1 \to^n T_2$ is the type of a function taking arguments of type T_1, returning a term of type T_2, and such that evaluation of the body accesses regions in the memory *lower than* the region n.

Stratification. We impose a well-formedness condition on types that reflects the stratification of the store: a term acting at region n cannot be stored in a region smaller than $n + 1$. For this, we define $\mathrm{reg}(T)$, an integer describing the set of regions associated to a type T, by:

$$
\mathrm{reg}(\mathbb{1}) = 0 \qquad \mathrm{reg}(T\ \mathtt{ref}_n) = \max(n, \mathrm{reg}(T))
$$
$$
\mathrm{reg}(T_1 \to^n T_2) = \max(n, \mathrm{reg}(T_2))
$$

Definition 1 (Well-formed types) *A type T is well-formed if for all its subtypes of the form T' $\mathtt{ref}_n$, we have* $\mathrm{reg}(T') < n$.

In the following, we shall implicitly assume that all types we manipulate are well-formed. Well-formedness of types is the condition that ensures the termination of the imperative part of a term. This in particular ensures that each time we reduce a redex $\mathtt{deref}_n(u_{(n,T)})$, the obtained value does not create new operations acting at region n.

Comparison with [3]. The type system we present in the next section is actually very close to the one given in [3], which in turn is close to the one of [1].

In our presentation, regions, defined in [3] as abstract parts of the store, are denoted by natural numbers. The two presentations are equivalent. In [3], when the stratification condition (which is inductively defined on sets of regions) is met, a partial order between regions can be extracted, and thus integers can be assigned to regions so that each typable term can be given a well-formed type using our definitions. Conversely, from a set of regions indexed by natural numbers we can derive easily a set of corresponding abstract regions satisfying the stratification condition.

Another difference between the two settings is that our well-formedness condition for types is actually looser than the one found in [3], allowing us to typecheck more terms. Indeed, in Definition 1, in the case of an arrow type, we do not impose the well-formedness condition in the type of the argument, making terms like $(\lambda x.(\mathtt{deref}_2(x)\ u_{(3,\mathbb{1})}))\ (\mathtt{ref}_2\ \lambda y.\star)$ acceptable in our setting, while they are not in [3]. In this example, x has type $\mathbb{1}\ \mathtt{ref}_3 \to^0 \mathbb{1}$ (detailed typing rules can be found in the following section), which gives type $(\mathbb{1}\ \mathtt{ref}_3 \to^0 \mathbb{1})\ \mathtt{ref}_2$ for $\mathtt{ref}_2\ \lambda y.\star$. Note that this example is phrased using natural numbers for regions: it is not difficult to translate it into Boudol's framework, and insert the term in an appropriate context in order to enforce that the (abstract) region corresponding to 3 dominates the region corresponding to 2.

We think that the works [3] and [1] can easily be adapted with this small refinement in our definition of well-formedness in order to obtain the same expressiveness as our system.

2.2 Types and Reduction

Typing. Figure 2 defines two typing judgements, of the form $\Gamma \vdash M : (T, n)$ for terms and $\Gamma \vdash \delta$ for stores. Our type system is presented *à la Church*, and we write $\Gamma(x) = T$ when variable x has type T according to type environment Γ.

In a typing judgement $\Gamma \vdash M : (T, n)$, n defines a bound on the *effect* of the evaluation of M, which intuitively corresponds to the highest region accessed when evaluating M. Effects can be thought of as sets of regions (the part of the store manipulated by the evaluation of a term), and are denoted by a single natural number, which stands for the maximum region in the effect.

As explained above, in type $T_1 \to^n T_2$, n refers to the effect of the body of the function. As a consequence, in rule (**App**), the effect of the application

Typing rules for terms

$$(\textbf{App})\ \frac{\Gamma \vdash M : (T_1 \to^n T_2, m) \qquad \Gamma \vdash N : (T_1, k)}{\Gamma \vdash M\ N : (T_2, \max(m,n,k))}$$

$$(\textbf{Abs})\ \frac{\Gamma \vdash M : (T_2, n) \qquad \Gamma(x) = T_1}{\Gamma \vdash \lambda x.\, M : (T_1 \to^n T_2, 0)}$$

$$(\textbf{Ref})\ \frac{\Gamma \vdash M : (T_1, m)}{\Gamma \vdash \mathtt{ref}_n\ M : (T_1\ \mathtt{ref}_n, \max(n,m))} \qquad (\textbf{Var})\ \frac{\Gamma(x) = T_1}{\Gamma \vdash x : (T_1, 0)}$$

$$(\textbf{Uni})\frac{}{\Gamma \vdash \star : (\mathbb{1}, 0)} \qquad (\textbf{Add})\ \frac{}{\Gamma \vdash u_{(n,T_1)} : (T_1\ \mathtt{ref}_n, 0)}$$

$$(\textbf{Asg})\ \frac{\Gamma \vdash M : (T_1\ \mathtt{ref}_n, m) \qquad \Gamma \vdash N : (T_1, k)}{\Gamma \vdash M \mathtt{:=}_n N : (\mathbb{1}, \max(m,n,k))}$$

$$(\textbf{Drf})\ \frac{\Gamma \vdash M : (T\ \mathtt{ref}_n, m)}{\Gamma \vdash \mathtt{deref}_n(M) : (T, \max(m,n))}$$

Typing rules for stores

$$(\textbf{Emp})\frac{}{\Gamma \vdash \emptyset} \qquad (\textbf{Sto})\frac{\Gamma \vdash \delta \qquad \Gamma \vdash V : (T, 0)}{\Gamma \vdash \delta\langle u_{(n,T)} \rightsquigarrow V\rangle}$$

Fig. 2. $\lambda_{\mathtt{ref}}$: Type and Effect System

$M\ N$ where M has type $T_1 \to^n T_2$ is the maximum between the effect of M, the effect of N, and n. Indeed the maximum region accessed during the evaluation of $M\ N$ is accessed during either the evaluation of M to some function $\lambda x.M_2$, or the evaluation of N to some value V_1, or during the evaluation of $M_2\{V_1/x\}$, whose effect is n.

We notice that values have an effect 0: values cannot reduce and, as explained above, the effect of a term stands for the maximum region accessed during its evaluation.

We extend typing to evaluation contexts by treating the hole as a term variable which can be given any type and has effect 0.

Reduction. The execution of programs is given by a reduction relation, written $\mapsto$, relating *states* (a state is given by a pair consisting of a term and a store), and which is defined on Figure 3. We write $\mapsto^n_{\mathrm{F}}$ for a *functional* reduction, obtained using rule (β); n refers to the effect of the β-redex, that is, in this call-by-value setting, the region that decorates the type of the function being triggered. In other words, we suppose in rule (β) that $\Gamma \vdash \lambda x.\, M : (T_V \to^n T, m)$ holds for some T_V, T, m. We introduce similarly *imperative* reductions, noted $\mapsto^n_{\mathrm{I}}$, for reductions obtained using rules (**ref**), (**deref**) or (**store**) (in these cases, the

$$(\beta)\frac{}{(\lambda x.\, M\ V, \delta) \mapsto (M\{V/x\}, \delta)}$$

$$(\mathbf{ref})\frac{u_{(n,T)} \notin \mathrm{supp}(\delta) \qquad \Gamma \vdash V : (T, \_)}{(\mathtt{ref}_n\ V, \delta) \mapsto (u_{(n,T)},\ \delta\langle u_{(n,T)} \rightsquigarrow V\rangle)}$$

$$(\mathbf{deref})\frac{\delta(u_{(n,T)}) = V}{(\mathtt{deref}_n(u_{(n,T)}), \delta) \mapsto (V, \delta)}$$

$$(\mathbf{store})\frac{\Gamma \vdash V : (T, \_)}{(u_{(n,T)} \mathtt{:=}_n V, (\delta)) \mapsto (\star,\ \delta\langle u_{(n,T)} \rightsquigarrow V\rangle)}$$

$$(\mathbf{context})\frac{(M, \delta) \mapsto (M', \delta')}{(\mathbf{E}[M], \delta) \mapsto (\mathbf{E}[M'], \delta')}$$

Fig. 3. $\lambda_{\mathtt{ref}}$: Reduction Rules

accessed region n appears explicitly in the rules of Figure 3). We will call a reduction according to $\mapsto^n_{\mathrm{F}}$ (resp. $\mapsto^n_{\mathrm{I}}$) "a functional reduction on level n" (resp. "an imperative reduction on level n").

Definition 2 *We define an* infinite computation starting from M *as an infinite sequence* $(M_i, \delta_i)_{0 \leq i}$ *such that* $M_0 = M$, $\delta_0 = \emptyset$ *and* $\forall i, (M_i, \delta_i) \mapsto (M_{i+1}, \delta_{i+1})$.

We say that a term M diverges *when there exists an infinite sequence starting from* M *and that* M terminates *when it does not diverge.*

The following result will be useful to prove Proposition 5. It says that we can replace a term inside an evaluation context with a term of the same type but with a smaller effect, while preserving typability. The effect of the whole term can decrease (in the case where $\mathbf{E} = [\,]$ for instance).

Lemma 3 *If* $\begin{cases} \Gamma \vdash \mathbf{E}[M] : (T, n) \\ \Gamma \vdash M : (T_0, m) \\ \Gamma \vdash M' : (T_0, m') \\ m' \leq m \end{cases}$ *then* $\Gamma \vdash \mathbf{E}[M'] : (T, n')$ *with* $n' \leq n$.

Our type and effect system enjoys the two standard properties of subject substitution and subject reduction. Notice that in the statement of Lemma 4, the effect associated to $M\{V/x\}$ is the same as the one associated to M. This holds as the term V is a value and thus does not introduce new operations on the memory which are not handled by the type system. Should we have used a call-by-name setting, the statement of this proposition would have been: "If $\Gamma \vdash M : (T, n)$, $\Gamma(x) = T'$ and $\Gamma \vdash N : (T', m)$ then $\Gamma \vdash M\{N/x\} : (T, \max(m, n))$".

Lemma 4 (Subject substitution)
If $\Gamma \vdash M : (T, n)$, $\Gamma(x) = T'$ *and* $\Gamma \vdash V : (T', m)$ *then* $\Gamma \vdash M\{V/x\} : (T, n)$.

We only sketch proofs for some results. The proof for Lemma 4, as well as detailed proofs for all other results, can be found in [5].

Proposition 5 (Subject reduction)
$\Gamma \vdash M : (T, n)$, $\Gamma \vdash \delta$ and $(M, \delta) \mapsto (M', \delta')$ entail that $\Gamma \vdash \delta'$ and $\Gamma \vdash M' : (T, n')$ for some $n' \leq n$.

Proof (Sketch). The proof is done by induction on the derivation of $(M, \delta) \mapsto (M', \delta')$. If the rule (**context**) is used, we rely on Lemma 3. If the rule (**beta**) is used, we use Lemma 4. Cases (**ref**) and (**store**) are easy. Case (**deref**) is done using the hypothesis that δ is well-typed.

3 Termination of $\lambda_{\mathtt{ref}}$ Programs

3.1 Defining a projection from $\lambda_{\mathtt{ref}}$ to $\lambda_{\mathbf{ST}}$

The technique of projection and simulation works as follows. First, we define a projection function, parametrised upon a region p (we will refer to a "projection on level p"), which strips a $\lambda_{\mathtt{ref}}$ term from its imperative constructs (and some of its functional parts), in order to obtain a $\lambda_{\mathbf{ST}}$ term.

Then, we prove a simulation result (Lemma 14 below), stating that when a well-typed state (M, δ) reduces to (M', δ') by a functional reduction on level p, the projection on level p of M reduces in at least one step to the projection on level p of M'; moreover, when (M, δ) reduces to (M', δ') by another type of reduction then either the projections on level p of M and M' are equal, or the projection of M reduces in at least one step to the projection of M'. This result is what makes the projection function divergence preserving, as announced in Section 1.

With these results at hand, we suppose, toward a contradiction, the existence of a diverging process M_0, and we show the existence of a region p such that an infinite computation starting from M_0 contains an infinite number of functional reductions on level p. Using the simulation lemma, we obtain by projection a diverging $\lambda_{\mathbf{ST}}$ term (as a functional reduction on level p is mapped to at least one step of reduction), which contradicts strong normalisation of $\lambda_{\mathbf{ST}}$.

Before turning to the formal definition of the projection function, let us explain informally how it acts on $\mathtt{deref}_n(M)$ — we already gave some ideas about the projection of $\mathtt{ref}_n\ M$ in Section 1. Again, the purpose of the projection is to remove the imperative command. Because we cannot just throw away M (this would invalidate the simulation lemma), we apply the projection function recursively to M. Once the projected version of M is executed, we replace the result with a value of the appropriate type, which we call a *generic value*.

More precisely, generic values are canonical terms that are used to replace a given subterm *once we know that no divergence can arise due to the evaluation of the subterm* (this would correspond either to a divergence of the subterm, or to a contribution to a more general divergence). They are defined as follows:

Definition 6 *Given a type T without the* `ref` *construct, the* generic value $\mathtt{V}_T$ of type T *is defined by:* $\mathtt{V}_{T\ \mathtt{ref}_n} = \mathtt{V}_{\mathbb{1}} = \star$, *and* $\mathtt{V}_{T_1 \to^n T_2} = \lambda x.\mathtt{V}_{T_2}$ *(*x *being of type* T_1 *in the latter term).*

In order to program the evaluation of a projected subterm and its replacement with a generic value, the definition of projection makes use of the following (families of) projectors:

$$\Pi^{(1,2)} = \lambda x.\lambda y.\, x \qquad\qquad \Pi^{(1,3)} = \lambda x.\lambda y.\lambda z.\, x \ .$$

In the following, we shall use these projectors in a well-typed fashion (that is, we pick the appropriate instance in the corresponding family).

In order to present the definition of the projection function, we need a last notion, that conveys the intuition that a given term M can be involved in a reduction on level p. This can be the case for two reasons. Either M is able to perform (maybe after some preliminary reduction steps) a reduction on level p, in which case, by the typing rules, the effect of M is greater than p, or M is a function that can receive some arguments and eventually perform a reduction on level p, in which case the type system ensures that its type T satisfies $\mathrm{reg}(T) \geq p$.

Definition 7 *Suppose* $\Gamma \vdash M : (T, n)$. *We say that* M *is* related to p *if either* $n \geq p$ *or* $\mathrm{reg}(T) \geq p$. *In the former (resp. latter) case, we say that* M *is related to* p *via its effect (resp. via its type).*

We extend this notion to evaluation contexts by treating the hole like a term variable, for a given typing derivation for a context (this is useful in particular in the statement of Lemma 13).

Notice that a term containing a subterm whose effect is p is not necessarily related to p: for instance, we can derive $\Gamma \vdash (\lambda x.\star)\ \lambda y.\mathtt{deref}_3(u_{(3,\mathbb{1})}) : (\mathbb{1}, 0)$ for an appropriate Γ, but this term is not related to 3, although we can derive $\Gamma' \vdash \mathtt{deref}_3(u_{(3,\mathbb{1})}) : (\mathbb{1}, 3)$ for some Γ' — one can easily check that this term cannot be used to trigger a reduction on level 3.

Definition 8 *Given a typable* M *of type* T, *we define the* projection on level p of M, *written* $\mathsf{pr}^p_\Gamma(M)$, *as follows:*

If M is not related to p:

$$\mathsf{pr}^p_\Gamma(M) = \mathtt{V}_T$$

Otherwise:

$$\begin{aligned}
\mathsf{pr}^p_\Gamma(M_1\ M_2) &= \mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(M_2)\\
\mathsf{pr}^p_\Gamma(x) &= x\\
\mathsf{pr}^p_\Gamma(\lambda x.M_1) &= \lambda x.\mathsf{pr}^p_\Gamma(M_1)\\
\mathsf{pr}^p_\Gamma(\mathtt{ref}_n\ M_1) &= (\Pi^{(1,2)}\ \star\ \mathsf{pr}^p_\Gamma(M_1))\\
\mathsf{pr}^p_\Gamma(\mathtt{deref}_n(M_1)) &= (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(M_1))\\
\mathsf{pr}^p_\Gamma(M_1 \mathtt{:=}_n M_2) &= (\Pi^{(1,3)}\ \star\ \mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(M_2))\\
\mathsf{pr}^p_\Gamma(u_{(n,T_1)}) &= \star
\end{aligned}$$

We extend this definition to evaluation contexts in the following way: we always propagate the projection inductively in a context $\mathbf{E}$, without checking if the context is related to p or not. For instance, $\mathsf{pr}^p_\Gamma(\mathbf{E}_1\ M) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)\ \mathsf{pr}^p_\Gamma(M)$ even if $(\mathbf{E}_1\ M)$ is not related to p.

The projection function maps $\lambda_{\mathtt{ref}}$ terms to $\lambda_{\mathbf{ST}}$ terms, where $\lambda_{\mathbf{ST}}$ is the simply typed λ-calculus: this is stated in Lemma 10.

Definition 9 *We extend the projection function to act on types as follows:*

$$\mathsf{pr}^p_\Gamma(\mathbb{1}) = \mathbb{1} \qquad \mathsf{pr}^p_\Gamma(T\ \mathtt{ref}_n) = \mathbb{1} \qquad \mathsf{pr}^p_\Gamma(T_1 \to^n T_2) = \mathsf{pr}^p_\Gamma(T_1) \to \mathsf{pr}^p_\Gamma(T_2)\ .$$

Observe that for any type T, $\mathsf{pr}^p_\Gamma(T)$ is a simple type, and $\mathtt{V}_T$ is a simply-typed λ-term of type $\mathsf{pr}^p_\Gamma(T)$.

Lemma 10 *Take $p \in \mathbb{N}$, and suppose $\Gamma \vdash M : (T, n)$. Then $\mathsf{pr}^p_\Gamma(M)$ belongs to $\lambda_{\mathbf{ST}}$, and has type $\mathsf{pr}^p_\Gamma(T)$.*

Proof (Sketch). We reason by induction on the typing judgement in $\lambda_{\mathtt{ref}}$. If M is not related to p, the result follows directly from the remarks above. Otherwise, we reason by cases on the last rule used to type M and conclude using the induction hypothesis.

3.2 Simulation Result

In order to reason about the transitions of projected terms, the first step is to understand how projection interacts with the decomposition of a term into an evaluation context and a redex.

The lemma below explains how the projection function is propagated within a term of the form $\mathbf{E}[M]$. There are, intuitively, two possibilities, depending only on the context and on the level (p) of the projection:

- either $\mathbf{E}$ is such that $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$ for all M, that is, the projection is always propagated in the hole to M,
- or this is not the case and the context is such that, if the effect of M is too small, the projection inserts a generic value before reaching the hole in $\mathbf{E}$. In this case $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[V]$, where $\mathbf{E}_1$ is an 'initial part' of $\mathbf{E}$, and this equality holds independently from M (as long as, like said above, the effect of M is sufficiently small in some sense).

In the former case, the projection is propagated inductively inside the context to the hole, no matter the effect of M, whereas in the latter case, if the effect of M is small enough, the projection does not stop before reaching the hole in $\mathbf{E}$.

Lemma 11 *Take $p \in \mathbb{N}$, and consider a well-typed context $\mathbf{E}$. We have:*

1. *Either for all well-typed process M, $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$,*
2. *or there exist $\mathbf{E}_1$ and $\mathbf{E}_2 \neq []$ s.t. $\mathbf{E} = \mathbf{E}_1[\mathbf{E}_2]$ and, for all M, if k stands for the effect of M, we are in one of the two following cases:*

(a) If $k \geq p$, then $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$.
(b) If $k < p$, then $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ (where T'' is the type of $\mathbf{E}_2$).

Proof (Sketch). We proceed by structural induction on $\mathbf{E}$ and distinguish two cases:

1. Either the context is not related to p. This means that $\mathbf{E}_1 = [\,]$ and $\mathbf{E}_2 = \mathbf{E}$. If $k < p$ then the projection of the whole term returns a generic value. If $k \geq p$ then we discuss on the structure of $\mathbf{E}$, use the induction hypothesis and the definition of projection.
2. If the context is related to p we discuss on the structure of the context and use the induction hypothesis, constructing at each step the outer context $\mathbf{E}_1$. When we reach a context not related to p, we conclude using case 1.

The properties we now establish correspond to the situation, in the previous lemma, where M is an imperative redex acting on region p. The typing rules of Figure 2 insure that firing the redex yields a term which is not related to p via its effect: depending on the kind of imperative operator that is executed, this term might either be related to p via its type, or not related to p at all.

In the latter case, we are able to show that the projected versions of the two terms are related by $\twoheadrightarrow^+$ (the transitive closure of reduction in $\lambda_{\mathbf{ST}}$), which allows us to establish a simulation property.

Fact 12 *If $\mathbf{E}_2$ is not related to p, then:*

1. *If $\mathbf{E}_2 = (V_3\ \mathbf{E}_3)$ then V_3 is not related to p.*
2. *If $\mathbf{E}_2 = (\mathbf{E}_3\ M_3)$ then $\mathbf{E}_3$ is not related to p.*

Lemma 13 *If $\Gamma \vdash \mathbf{E}_2 : (T'', m)$ and $\mathbf{E}_2$ is not related to p, then for any well-typed M, M',*

1. *$\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ \mathtt{V}_T\ M)] \twoheadrightarrow^+ \mathtt{V}_{T''}$;*
2. *$\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,3)}\ \mathtt{V}_T\ M\ M')] \twoheadrightarrow^+ \mathtt{V}_{T''}$.*

Proof (Sketch). We proceed by structural induction on $\mathbf{E}_2$. Fact 12 is necessary: for instance, if $\mathbf{E}_2 = \mathbf{E}_3\ M_3$, we have

$$\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)] = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)]\ \mathsf{pr}^p_\Gamma(M_3))$$

with $\mathbf{E}_3$ of type $T_3 \to T''$. Thus, we can use Fact 12 and the induction hypothesis on $\mathbf{E}_3$ to get $\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)] \twoheadrightarrow^+ \mathtt{V}_{T_3 \to T''}$, from which we conclude.

Lemmas 11 and 13 allow us to derive the desired simulation property for $\lambda_{\mathtt{ref}}$, the main point being that a functional reduction on level p is projected into one reduction in the target calculus (case 4 below).

Lemma 14 (Simulation) *Consider $p \in \mathbb{N}$, and suppose $\Gamma \vdash M : (T, m)$.*

1. *If $(M, \delta) \mapsto^n_{\mathrm{I}} (M', \delta')$ and $n < p$, then $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$.*

2. *If* $(M,\delta) \mapsto_{\mathrm{I}}^{p} (M',\delta')$, *then* $\mathsf{pr}_{\Gamma}^{p}(M) \twoheadrightarrow^{+} \mathsf{pr}_{\Gamma}^{p}(M')$.
3. *If* $(M,\delta) \mapsto_{\mathrm{F}}^{n} (M',\delta')$ *and* $n < p$, *then* $\mathsf{pr}_{\Gamma}^{p}(M) = \mathsf{pr}_{\Gamma}^{p}(M')$.
4. *If* $(M,\delta) \mapsto_{\mathrm{F}}^{p} (M',\delta')$, *then* $\mathsf{pr}_{\Gamma}^{p}(M) \twoheadrightarrow \mathsf{pr}_{\Gamma}^{p}(M')$.

Proof (Sketch). The structure of the proof is as follows. For cases 1 and 2, terms are decomposed in the same way but the arguments invoked are different. In case 1, we use the definition of projection on terms not related to p to conclude; in case 2, projection yields an "actual term" (not a generic value) and we use Lemma 13 to conclude.

In these reasonings, the proofs for rules (**ref**) and (**deref**) differ, as in the former case the more complex term appears before the reduction (we have $\mathtt{ref}_n\ V$ which reduces to $u_{(n,T)}$) whereas in the latter case the more complex term appears after the reduction (we have $\mathtt{deref}_n(u_{(n,T)})$ which reduces to V).

Cases 3 and 4 are treated along the lines of cases 1 and 2, except that Lemma 13 is not required.

3.3 Deriving soundness

To obtain soundness, we need to show that a diverging term performs an infinite number of functional reductions on level p, for some p. For this we introduce a measure that decreases along imperative reductions on level p and does not increase along reductions on level $< p$. The measure is given by counting the *active imperative operators* of a term, which are the imperative operators (reference creations, dereferencings and assignments) that do not occur under a λ.

Definition 15 *Take M in $\lambda_{\mathtt{ref}}$. The number of active imperative operators on region p in M, written $\mathbf{Ao}^p(M)$ is defined inductively as follows:*

$$\mathbf{Ao}^p(x) = \mathbf{Ao}^p(\lambda x.M) = \mathbf{Ao}^p(u_{(n,T)}) = 0 \qquad \mathbf{Ao}^p(M\ N) = \mathbf{Ao}^p(M) + \mathbf{Ao}^p(N)$$

$$\begin{array}{l} \mathbf{Ao}^p(\mathtt{deref}_n(M)) = \mathbf{Ao}^p(\mathtt{ref}_n\ M) = \mathbf{Ao}^p(M) \qquad \text{if } n \neq p \\ \mathbf{Ao}^p(\mathtt{deref}_p(M)) = \mathbf{Ao}^p(\mathtt{ref}_p\ M) = 1 + \mathbf{Ao}^p(M) \end{array}$$

$$\begin{array}{l} \mathbf{Ao}^p(M :=_n N) = \mathbf{Ao}^p(M) + \mathbf{Ao}^p(N) \qquad \text{if } n \neq p \\ \mathbf{Ao}^p(M :=_p N) = 1 + \mathbf{Ao}^p(M) + \mathbf{Ao}^p(N) \end{array}$$

$\mathbf{Ao}^p(M)$ and the effect of M are related as follows:

Lemma 16 *If $\Gamma \vdash M : (T,m)$ and $m < p$ then $\mathbf{Ao}^p(M) = 0$.*

We are finally able to show that $\mathbf{Ao}^p(M)$ yields the measure we need.

Lemma 17 *If $\Gamma \vdash M : (T,m)$ then:*

1. *if $(M,\delta) \mapsto_{\mathrm{F}}^{n} (M',\delta')$ with $n < p$ then $\mathbf{Ao}^p(M') \leq \mathbf{Ao}^p(M)$,*
2. *if $(M,\delta) \mapsto_{\mathrm{I}}^{n} (M',\delta')$ with $n < p$ then $\mathbf{Ao}^p(M') \leq \mathbf{Ao}^p(M)$,*
3. *and if $(M,\delta) \mapsto_{\mathrm{I}}^{p} (M',\delta')$ then $\mathbf{Ao}^p(M') < \mathbf{Ao}^p(M)$.*

Proof (Sketch). We reason by cases on the reduction rules and use Lemma 16 to show that new imperative operators on region p can only be generated by functional reductions on level $\geq p$ or by imperative reductions on level $> p$, and that each imperative reduction on level p erases one active imperative operator on region p.

The following lemma states that there exists a maximum region p on which an infinite number of reductions takes place. With the previous result, we can deduce that an infinite number of functional reductions take place on level p.

Lemma 18 *Suppose that $\Gamma \vdash M : (T, l)$, and that there exists $(M_i, \delta_i)_{i\in\mathbb{N}}$, an infinite reduction sequence starting from M. Then:*

1. *For all i, M_i is typable.*
2. *There exist p and i_o s.t.*
   (a) *if $i > i_0$ and $(M_i, \delta_i) \mapsto_{\mathrm{I}}^{n} (M_{i+1}, \delta_{i+1})$ then $n \leq p$,*
   (b) *if $i > i_0$ and $(M_i, \delta_i) \mapsto_{\mathrm{F}}^{n} (M_{i+1}, \delta_{i+1})$ then $n \leq p$,*
   (c) *There exists an infinite set of indexes $\mathcal{I}$ s.t. for each $i \in \mathcal{I}$, either $(M_i, \delta_i) \mapsto_{\mathrm{F}}^{p} (M_{i+1}, \delta_{i+1})$ or $(M_i, \delta_i) \mapsto_{\mathrm{I}}^{p} (M_{i+1}, \delta_{i+1})$.*
   (d) *There are infinitely many $i \in \mathcal{I}$ s.t. $(M_i, \delta_i) \mapsto_{\mathrm{F}}^{p} (M_{i+1}, \delta_{i+1})$.*

Proof (Sketch).

1. Follows from Proposition 5.
2. The set of different regions is finite, so we easily find a p satisfying 2a, 2b and 2c. Lemma 17 ensures that 2d holds.

Theorem 19 (Soundness) *If $\Gamma \vdash M : (T, m)$ then M terminates.*

Proof. Consider, by absurd, an infinite computation $(M_i, \delta_i)_i$ starting from $M = M_0$ and δ_0. By Lemma 18, all the M_i's are well-typed, and there is a maximal p s.t. for infinitely many i, $(M_i, \delta_i) \mapsto_{\mathrm{F}}^{p} (M_{i+1}, \delta_{i+1})$. Furthermore, there exists i_0 such that every reduction on an index greater than i_0 is performed on region $n \leq p$. Consider the sequence $(\mathsf{pr}_\Gamma^p(M_i))_{i>i_0}$. By Lemma 14, we obtain that for every $i > i_0$, $\mathsf{pr}_\Gamma^p(M_i) \twoheadrightarrow^* \mathsf{pr}_\Gamma^p(M_{i+1})$. Moreover, $\mathsf{pr}_\Gamma^p(M_i) \twoheadrightarrow^+ \mathsf{pr}_\Gamma^p(M_{i+1})$ for an infinite number of i. Thus $\mathsf{pr}_\Gamma^p(M_{i_0})$ is diverging. This contradicts the termination of $\lambda_{\mathbf{ST}}$.

Remark 20 (Raising the effect) *The results we present in this paper still hold if we add the rule:*

$$(\mathbf{Sub})\frac{\Gamma \vdash M : (T, n) \qquad n \leq n'}{\Gamma \vdash M : (T, n')}$$

to the type system.

This rule allows us to be more liberal when typing terms, thus obtaining a greater expressiveness. For instance, it allows one to store at the same address functions whose bodies do not have the same effect.

Example 21 (Landin's trick) *The standard example of diverging term in* λ_{ref}*, known as* Landin's trick*, is given by:*

$$(\lambda f.[(\lambda t.(\mathtt{deref}_1(f)\ \star))\ (f :=_1 \lambda z.(\mathtt{deref}_1(f)\ z))])\ (\mathtt{ref}_1\ \lambda x.x)\ .$$

In order to try and type this term, we are bound to manipulate non well-formed types.

In the call by value setting of λ_{ref}*, a first address* $u_{(1,\mathbb{1}\to^1\mathbb{1})}$ *(we use Remark 20 here, as the identity has no effect) is created when evaluating the argument (*$\mathtt{ref}_1\ \lambda x.x$*); this address instantiates* f *in the body of the outer function. Then* $u_{(1,\mathbb{1}\to^1\mathbb{1})}$ *is updated using the function* $\lambda z.(\mathtt{deref}_1(f)\ z)$*, whose type is* $\mathbb{1} \to^1 \mathbb{1}$*, at which point the term enters a loop. It is easy to see that the type of* $u_{(1,\mathbb{1}\to^1\mathbb{1})}$ *(which is also the type of* f*) is* $(\mathbb{1} \to^1 \mathbb{1})\ \mathtt{ref}_1$ *and is not well-formed, as* $\mathrm{reg}(\mathbb{1} \to^1 \mathbb{1}) = 1 \not\leq 1$*.*

On the other hand, consider the following terminating term:

$$(\lambda f.[(\lambda t.(\mathtt{deref}_1(f)\ \star))\ (f :=_1 \lambda z.(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))\ z])\ (\mathtt{ref}_1\ \lambda x.x)$$

where $I = \lambda t.\, t$*. This term is close to the example given above, except that* $\lambda z.(\mathtt{deref}_1(f)\ z)$ *is replaced with* $\lambda z.(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y)\ z$*. This new subterm, stored at address* f*, contains a dereferencing of* f*. Yet the term terminates because the dereferencing never comes in redex position. Indeed, the term* $(\lambda z.(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))\ z)$ *reduces to* $(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))$ *which, in turn, reduces in two steps to* I*.*

Here the type system assigns to $(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))$ *the type* $\mathbb{1} \to^0 \mathbb{1}$ *and the effect* 0*. Thus the type of* x *is* $\mathbb{1} \to^0 \mathbb{1}\ \mathtt{ref}_1$*, which is well-formed.*

4 Parametricity

As is the case in [7] for the π-calculus, the method we have presented for the λ-calculus with references is parametric with respect to a terminating purely functional core, and does not examine the corresponding termination proof. Other core calculi could be considered. Moreover, if the functional calculus corresponds to a subset of the simply typed terms, then the result holds directly.

We believe that it is possible to extend our work to polymorphic types, although this extension is not trivial if we consider adding region polymorphism: for instance, we would have to guarantee that a type like $(\forall A.A \to^0 A)\ \mathtt{ref}_n$ cannot have its A component instantiated with a type containing a region strictly greater than n.

Another idea is to apply this termination technique to a language containing both references and a recursion operator on integers. By restricting the use of the latter (in order not to create loops based on recursion), we think that one could be able to enrich the system we have presented.

By taking as functional core a λ-calculus with complexity bounds (such as, for instance, [2]), we believe that one can use our technique in order to lift complexity bounds for impure languages. The main idea is to rely on the projection function

to provide bounds on the number of reductions a terminating typed term can make.

Note, to conclude, that references can be encoded in a standard way in the π-calculus (as well as the call-by-value λ-calculus). One could then wonder if the method presented in [7] can recognise as terminating the subset of π-processes corresponding to encodings of $\lambda_{\mathtt{ref}}$ terms. The question is challenging, as weight-based methods for termination in π [8] cannot be used to prove termination of the encoding of $\lambda_{\mathbf{ST}}$ [6, 4].

Acknowledgements. Support from the french ANR projects "CHoCo", "AEOLUS" and "Complice" (ANR-08-BLANC-0211-01), and by the European Project "HATS" (contract number 231620) is acknowledged.

References

1. R. M. Amadio. On Stratified Regions. In *Proc. of APLAS*, 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. *CoRR*, abs/1005.0835, 2010.
3. G. Boudol. Fair Cooperative Multithreading. In *Proc. of CONCUR*, volume 4703 of *LNCS*, pages 272–286. Springer, 2007.
4. I. Cristescu and D. Hirschkoff. Termination in a π-calculus with Subptying. in preparation, 2011.
5. R. Demangeon. *Termination for Concurrent Systems.* PhD thesis, Ecole Normale Superieure de Lyon, 2010.
   Available from `http://perso.ens-lyon.fr/romain.demangeon/phd.pdf`.
6. R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Mobile Processes and Termination. In *Semantics and Algebraic Specification*, volume 5700 of *LNCS*, pages 250–273. Springer, 2009.
7. R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Termination in Impure Concurrent Languages. In *Proc. of CONCUR'10*, volume 6269 of *LNCS*, pages 328–342. Springer Verlag, 2010.
8. Y. Deng and D. Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7):1045–1082, 2006.
9. P. Tranquilli. Translating types and effects with state monads and linear logic. submitted, 2011.