Compilation of Extended Recursion in Call-by-Value Functional Languages *

Tom Hirschowitz
INRIA Rocquencourt
Tom.Hirschowitz@inria.fr

Xavier Leroy
INRIA Rocquencourt
Xavier.Leroy@inria.fr

J. B. Wells
Heriot-Watt University
jbw@macs.hw.ac.uk

ABSTRACT

This paper formalizes and proves correct a compilation scheme for mutually-recursive definitions in call-by-value functional languages. This scheme supports a wider range of recursive definitions than standard call-by-value recursive definitions. We formalize our technique as a translation scheme to a lambda-calculus featuring in-place update of memory blocks, and prove the translation to be faithful.

Categories and Subject Descriptors

F.3.3 [**Logics and meanings of programs**]: Studies of program constructs—*program and recursion schemes*; F.3.2 [**Logics and meanings of programs**]: Semantics of programming languages—*operational semantics*; D.3.1 [**Programming languages**]: Formal definitions and theory—*Syntax,semantics*; D.3.3 [**Programming languages**]: Language constructs and features—*Recursion*; D.3.3 [**Programming languages**]: Processors—*Compilers*

General Terms

Design, languages, reliability, theory

Keywords

compilation, recursion, semantics, functional languages

1. INTRODUCTION

Functional languages usually feature mutually recursive definition of values. In ML, this is supported by the let rec construct. Languages differ, however, in the kind of expressions they allow as right-hand sides of mutually recursive definitions. For instance, Haskell [7] allows arbitrary expressions as right-hand sides of recursive definitions, while Standard ML [13] only allows syntactic λ-abstractions, and OCaml [12, 11] allows both λ-abstractions and limited forms of constructor applications.

Several criteria come into play when determining the range of allowed right-hand sides. First, languages have to give a status to ill-founded definitions such as `x = x + 1`. In a lazy language, this definition can be represented by a recursive block of code. When its evaluation is requested, this code is executed, but it begins by requesting its own evaluation. So, depending on the compiler, it will either loop indefinitely or result in a run-time error. For call-by-value languages, ill-founded definitions are more problematic: during the evaluation of `x = x + 1`, the right-hand side `x + 1` must be evaluated while the value of `x` is still unknown. There is no strict call-by-value strategy that allows this. Thus, such ill-founded definitions must be rejected. Moreover, the burden recursive definitions impose to the rest of the compiler must be taken into account. For example, one could systematically implement recursive definitions through reference cells or thunks, but this would force the compiler to maintain information about whether values are recursive or not. Finally, the efficiency of the generated code is important. All these criteria interact tightly, yielding a tension between expressiveness, efficiency, and simplicity.

Recent work by Boudol [4] introduces a call-by-value let rec construct that is more expressive than that of ML or OCaml. In Boudol's work, right-hand sides of recursive definitions are not syntactically restricted, but ill-founded definitions are ruled out by a type system. This approach is further refined by Hirschowitz and Leroy [8]. Boudol and Zimmer [5] propose an implementation technique for this extended let rec, where recursive definitions of syntactic functions are implemented in a standard way, while reference cells are introduced to deal with more complex recursive definitions. The implementation of the Scheme `letrec` construct proposed by Waddell et al. [15] follows the same approach.

The present paper develops and proves correct a compilation scheme and call-by-value evaluation strategy for an extended let rec construct. This let rec construct supports both λ-abstractions and record constructions as right-hand sides of recursive definitions. Moreover, it allows non-recursive definitions to be interleaved with recursive definitions within a single let rec binding. The compilation scheme we propose for this flavor of let rec is a generalization of the "in-place update trick" described by Cousineau et al. [6]. It is less expressive than that of Boudol [5], as discussed in section 7,

*Partially supported by EPSRC grant GR/R 41545/01

PPDP'03, August 27–29, 2003, Uppsala, Sweden.

but it is simpler and more efficient, since it does not require the introduction of reference cells.

Our main motivation in studying this extended let rec construct is that it plays an important role in the language of call-by-value mixin modules currently investigated by the authors [10]. Moreover, the OCaml compiler uses a subset of the compilation scheme described here to compile non-functional recursive definitions; this paper is the first formal proof of the correctness of this compilation scheme.

The remainder of this paper is organized as follows. In section 2, we first review informally the "in-place update trick" [6], and show that it extends to combinations of recursive and non-recursive bindings within the same let rec. In section 3, we formalize the corresponding source language $\lambda_\circ$. Section 4 defines a target language λ_{alloc}, featuring in-place update of memory blocks. We define the compilation scheme from $\lambda_\circ$ to λ_{alloc} in section 5, and prove its correctness in section 6. Related work and conclusions are discussed in sections 7 and 8. Proofs are omitted in this paper, but can be found in a companion technical report [9].

2. THE IN-PLACE UPDATE TRICK

The original scheme The "in-place update trick" outlined by Cousineau et al. [6] and refined in the OCaml compiler [11], implements let rec definitions that satisfy the following two conditions. For any mutually recursive definition $x_1 = e_1 \ldots x_n = e_n$, first, the value of each definition should be represented at run-time by a heap allocated block of statically predictable size; second, for each i, the computation of e_i should not need the value of any of the definitions e_j, but only their names x_j. As an example of the second condition, the recursive definition `f = λx.(... f ...)` is accepted, since the computation of the right-hand side does not need the value of `f`. We say that it safely depends on `f`. In contrast, the recursive definition `f = (f 0)` is rejected. We say that the right-hand side strictly depends on `f`.

Evaluation of a let rec definition with in-place update consists of three steps. First, for each definition, allocate an uninitialized block of the expected size, and bind it to the recursively-defined identifier. Those blocks are called dummy blocks. Second, compute the right-hand sides of the definitions. Recursively-defined identifiers thus refer to the corresponding dummy blocks. Owing to the second condition, no attempt is made to access the contents of the dummy blocks. This step leads, for each definition, to a block of the expected size. Third, the contents of the obtained blocks are copied to the dummy blocks, updating them in place.

For example, consider a mutually recursive definition $x_1 = e_1, x_2 = e_2$, where it is statically predictable that the values of the expressions e_1 and e_2 will be represented at runtime by heap allocated blocks of sizes 2 and 1, respectively. Here is what the compiled code does, as depicted in figure 1. First, it allocates two uninitialized heap blocks, at addresses ℓ_1 and ℓ_2, of respective sizes 2 and 1. This is called the pre-allocation step. As a second step, it computes e_1, where x_1 and x_2 are bound to ℓ_1 and ℓ_2, respectively. The result is a heap block of size 2, with possible references to the two uninitialized blocks. The same process is carried on for e_2, resulting in a heap block of size 1. The third and final step copies the contents of the two obtained blocks to the two uninitialized blocks. The result is that the two initially dummy blocks now contain the proper cyclic data structures.

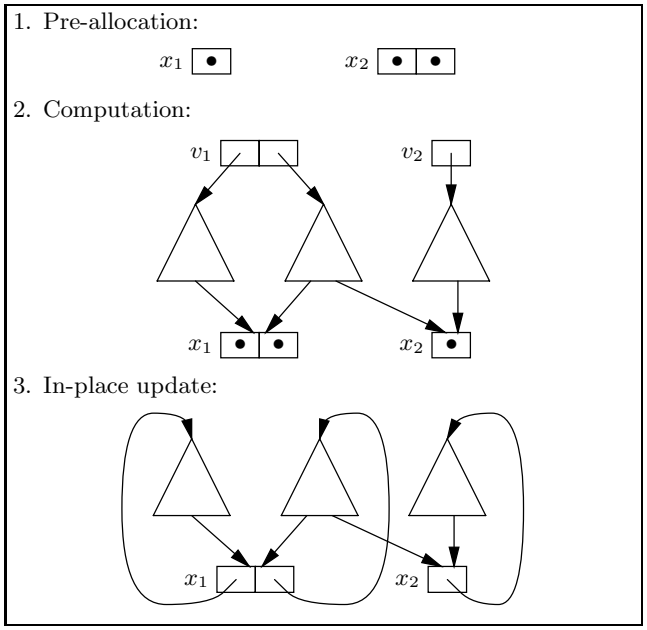


Figure 1: The in-place update trick

The extended scheme The scheme described above computes all definitions in sequence, and only then updates the dummy blocks in place. From the example above, it seems quite clear that in-place update for a definition could be done as soon as its value is available.

As long as definitions safely depend on each other, as happens with functions for instance, both schemes behave identically. Nevertheless, in the case where e_2 strictly depends on x_1, for example if $e_2 = \mathtt{fst}(x_1) + 1$, the original scheme can go wrong. Indeed, the contents of the dummy block pre-allocated for x_1 are still undefined when e_2 is computed. Instead, with immediate in-place update, the value v_1 is already available when computing e_2. This trivial modification to the scheme thus increases the expressive power of let rec. It allows definitions to de-structure the values of previous definitions. Furthermore, it allows to introduce definitions with unknown sizes in let rec, as shown by the following example.

An example of execution is presented in figure 2. The definition is $x_1 = e_1, x_2 = e_2, x_3 = e_3$, where e_1 and e_3 are expected to evaluate to blocks of sizes 2 and 1, respectively, but where the representation for the value of e_2 is not statically predictable. The pre-allocation step only allocates dummy blocks for x_1 and x_3. The value v_1 of e_1 is then computed. It can reference x_1 and x_3, which correspond to pointers to the dummy blocks, but not x_2, which would not make any sense here. This value is copied to the corresponding dummy block. Then, the value v_2 of e_2 is computed. The computation can refer to both dummy blocks, but it can also strictly depend on x_1. Finally, the value v_3 of e_3 is computed and copied to the corresponding dummy block.

This modified scheme implements more mutually recursive definitions than the initial one. The next section formalizes its semantics.

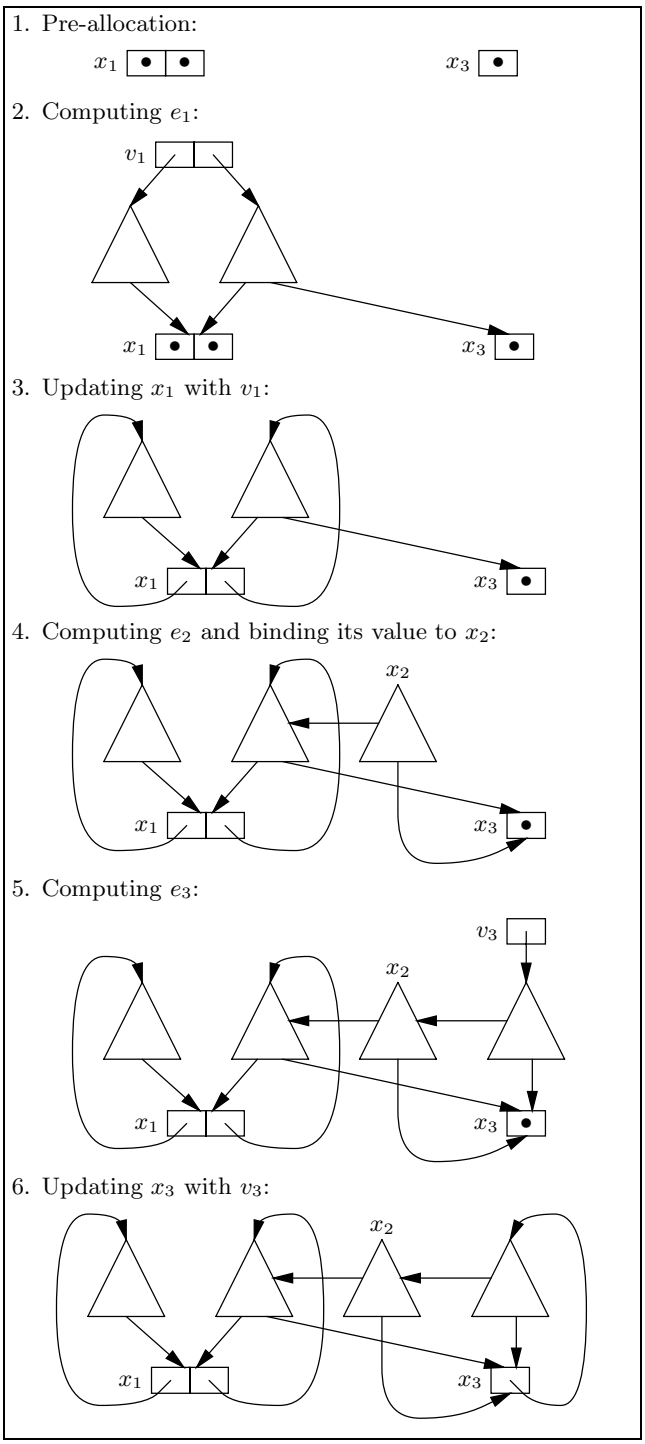


Figure 2: The refined in-place update trick

Variables	$x \in Vars$		
Names	$X \in Names$		
Expressions	$e \in expr$	::=	$x \mid \lambda x.e \mid e_1\ e_2$
		$\mid$	$\{s\} \mid e.X$
		$\mid$	let rec b in e
Records	s	::=	$X_1 = e_1 \ldots X_n = e_n$
Bindings	b	::=	$x_1 = e_1 \ldots x_n = e_n$
Expressions of predictable shape			
	$e_\downarrow \in Predictable$	::=	$\lambda x.e \mid \{s\}$
		$\mid$	let rec b in $e_\downarrow$

Figure 3: Syntax of $\lambda_\circ$

3. THE SOURCE LANGUAGE

3.1 Syntax

The syntax of $\lambda_\circ$ is defined in figure 3. The meta-variables X and x range over names and variables, respectively. Names are used for accessing record fields. The language includes λ-calculus: variables x, abstraction $\lambda x.e$, and application $e_1\ e_2$. The language also features records $\{X_1 = e_1 \ldots X_n = e_n\}$, record selection $e.X$ and a let rec construct. A mutually recursive definition has the shape let rec $x_1 = e_1 \ldots x_n = e_n$ in e.

Syntactic correctness Records $s = (X_1 = e_1 \ldots X_n = e_n)$ and bindings $b = (x_1 = e_1 \ldots x_n = e_n)$ are required to be finite maps: a record is a finite map from names to expressions, and a binding is a finite map from variables to expressions. Requiring them to be finite maps means that they should not bind the same variable or name twice.

In the sequel, we refer collectively to records and bindings as sequences, and use the usual notions on finite maps f, such as the domain $dom(f)$, the codomain $cod(f)$, the restriction $f_{|P}$ to a set P, or the co-restriction $f_{\setminus P}$ outside of a set P, which is the restriction to the set $dom(f) \setminus P$.

Syntactic correctness of let rec bindings includes an additional requirement on dependencies between definitions. In a let rec binding $b = (x_1 = e_1 \ldots x_n = e_n)$, we say that there is a backward dependency of x_i on x_j if $1 \le i \le j \le n$ and $x_j \in FV(e_i)$. This let rec binding is syntactically correct only if, for any backward dependency of x_i on x_j, the expression e_j is of predictable shape. An expression of predictable shape, written $e_\downarrow$, is either a function abstraction, a record, or a binding followed by an expression of predictable shape. (See figure 3.)

Structural equivalence We consider expressions equivalent up to alpha-conversion of variables bound in λ or let rec expressions. The set of terms of $\lambda_\circ$ is defined as the set of structural equivalence classes of syntactically correct expressions.

3.2 Semantics

The semantics of $\lambda_\circ$ is quite standard, except for the treatment of let rec bindings.

As shown in figure 5, values include function abstractions $\lambda x.e$ and records of values $\{s_v\}$, where s_v denotes an evaluated record $X_1 = v_1 \ldots X_n = v_n$. Notice that variables are also values. This is required to allow the reduction of recursive definitions of the form let rec $x = (\lambda y.e)\ x$.

The semantics of record selection and of function application are defined in figure 4, by computational contraction

Computational contraction rules

$$\{X_1 = v_1 \ldots X_n = v_n\}.X_i \rightsquigarrow_c v_i \quad \text{(PROJECT)} \qquad \frac{x \notin FV(v)}{(\lambda x.e)\ v \rightsquigarrow_c \mathsf{let\ rec}\ x = v\ \mathsf{in}\ e} \quad \text{(BETA)}$$

$$\frac{dom(b) \cap FV(\mathbb{L}) = \emptyset}{\mathbb{L}[\mathsf{let\ rec}\ b\ \mathsf{in}\ e] \rightsquigarrow_c \mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{L}[e]} \quad \text{(LIFT)}$$

Computational reduction rules

$$\frac{e \rightsquigarrow_c e'}{\mathbb{E}[e] \dashrightarrow_c \mathbb{E}[e']} \quad \text{(CONTEXT)} \qquad \frac{dom(b_1) \cap (\{x\} \cup dom(b_v, b_2) \cup FV(b_v, b_2) \cup FV(f)) = \emptyset}{(b_v, x = (\mathsf{let\ rec}\ b_1\ \mathsf{in}\ e), b_2 \vdash f) \dashrightarrow_c (b_v, b_1, x = e, b_2 \vdash f)} \quad \text{(IM)}$$

$$\frac{dom(b) \cap (dom(b_v) \cup FV(b_v)) = \emptyset}{(b_v \vdash \mathsf{let\ rec}\ b\ \mathsf{in}\ e) \dashrightarrow_c b_v, b \vdash e} \quad \text{(EM)} \qquad \frac{\mathbb{E}[\mathbb{D}](x) = v}{\mathbb{E}[\mathbb{D}[x]] \dashrightarrow_c \mathbb{E}[\mathbb{D}[v]]} \quad \text{(SUBST)}$$

Evaluation contexts

Lift context:
$\mathbb{L} ::= \square\ e \mid v\ \square \mid \square.X \mid \{\mathbb{S}\}$

Nested lift context:
$\mathbb{F} ::= \square \mid \mathbb{L}[\mathbb{F}]$

Evaluation context:
$\mathbb{E} ::= (b_v \vdash \mathbb{F}) \mid (\mathbb{B}[\mathbb{F}] \vdash e)$

Record contexts:
$\mathbb{S} ::= s_v, X = \square, s$

Binding contexts:
$\mathbb{B} ::= b_v, x = \square, b$

Dereferencing contexts:
$\mathbb{D} ::= \square\ v \mid \square.X$

Access in evaluation contexts

$$(b_v \vdash \mathbb{F})(x) = b_v(x) \quad \text{(EA)} \qquad (b_v, y = \mathbb{F}, b \vdash e)(x) = b_v(x) \quad \text{(IA)}$$

Figure 4: Reduction semantics for $\lambda_\circ$

Configurations	c	::=	$b \vdash e$
Values	$v \in values$	::=	$x \mid \lambda x.e \mid \{s_v\}$
Value records	s_v	::=	$X_1 = v_1 \ldots X_n = v_n$
Value bindings	b_v	::=	$x_1 = v_1 \ldots x_n = v_n$
Answers	$a \in answers$	::=	$b_v \vdash v$

Figure 5: Configurations and answers in $\lambda_\circ$

rules, defining the local computational contraction relation $\rightsquigarrow_c$. Record projection selects the appropriate field in the record. The application of a function $\lambda x.e$ to a value v reduces to the body of the function where the argument is bound to x using a let rec.

The remaining rules in figure 4 are computational reduction rules that deal with the reduction of let rec bindings. These rules implement a deterministic evaluation strategy over the five basic operations on recursive bindings identified by Ariola et al. [2]. Before explaining the strategy, we first recall these five basic operations.

1. let rec **lifting** lifts a let rec node up one level in an expression. For example, an expression of the shape $e_1 + (\mathsf{let\ rec}\ b\ \mathsf{in}\ e_2)$ becomes $\mathsf{let\ rec}\ b\ \mathsf{in}\ e_1 + e_2$.

2. **Internal merging**. During the evaluation of a binding, a definition may return a let rec as an answer, where a value is expected. Internal merging merges this binding with the current one. An expression of the shape $\mathsf{let\ rec}\ b_1, x = (\mathsf{let\ rec}\ b_2\ \mathsf{in}\ e), b_3\ \mathsf{in}\ f$ becomes $\mathsf{let\ rec}\ b_1, b_2, x = e, b_3\ \mathsf{in}\ f$, provided no variable capture occurs.

3. **External merging**. As shown in figure 5, the shape of answers in $\lambda_\circ$ allows only one binding to wrap values. Therefore, if evaluation results in two nested bindings, they must be merged into a single one. An expression of the shape $\mathsf{let\ rec}\ b_1\ \mathsf{in}\ \mathsf{let\ rec}\ b_2\ \mathsf{in}\ e$ becomes $\mathsf{let\ rec}\ b_1, b_2\ \mathsf{in}\ e$, provided no variable capture occurs.

4. **External substitution** allows to access bound variables that are defined by an enclosing binding. Given any context $\mathbb{C}$, an expression of the shape $\mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{C}[x]$ becomes $\mathsf{let\ rec}\, b\, \mathsf{in}\, \mathbb{C}[e]$, if $x = e$ appears in b, and x is not captured by $\mathbb{C}$, and no variable capture occurs.

5. **Internal substitution** allows to access identifiers bound earlier in the same binding. (Assuming left-to-right evaluation, "earlier" means "to the left of".) An expression of the shape $\mathsf{let\ rec}\ b_1, y = \mathbb{C}[x], b_2\ \mathsf{in}\ e$ becomes $\mathsf{let\ rec}\, b_1, y = \mathbb{C}[f], b_2\, \mathsf{in}\, e$ if $x = f$ appears in b_1, and x is not captured by $\mathbb{C}$, and no variable capture occurs.

The issue is how to arrange these operations to make the evaluation deterministic and ensure that it reaches the answer when it exists. Our choice can be summarized as follows. There is a top-most binding. When this top-most binding is already evaluated, evaluation can proceed under this binding. Otherwise, evaluation is allowed inside this

binding. If evaluation meets another binding inside the expression, this binding is lifted until it is immediately under the top-most binding. Then, it is merged with it, internally or externally according to the context. External and internal substitutions are allowed only from the evaluated part of the top-most binding, and when the substituted variable is in a dereferencing context (see below). In order to simplify the presentation of the translation and the correctness proof, we distinguish this top-most binding syntactically: the global computational reduction relation $\dashrightarrow_c$ is a binary relation on configurations c, which are pairs of a binding, the top-most binding, and an expression, written $b \vdash e$ (see figure 5). Thus, the top-most binding plays the role of an environment, with the additional feature that values bound in this environment can be mutually recursive.

More formally, the contraction rule LIFT lifts a let rec binding up a lift context $\mathbb{L}$. As defined in figure 4, a lift context is any non-let rec expression where the context hole $\Box$ appears immediately under the first node, in position of the next sub-expression to be evaluated.

The computational reduction relation $\dashrightarrow_c$ extends the computational contraction relation to any evaluation context $\mathbb{E}$, as defined in figure 4. A nested lift context $\mathbb{F}$ is a series of nested lift contexts, and an evaluation context $\mathbb{E}$ is a nested lift context, possibly inside the (partially evaluated) top-most binding, or under the (fully evaluated) top-most binding.

The reduction rule IM corresponds to internal merging. If, during the evaluation of the top-most binding, one definition evaluates to a binding, then this binding is merged with the top-most one, provided no variable capture occurs. The evaluation can then continue.

The EM reduction rule corresponds to external merging. It is only possible at top-level, provided no variable capture occurs.

Finally, the external and internal substitution operations are modeled within a single reduction rule SUBST. This rule transforms an expression of the shape $\mathbb{E}[\mathbb{D}[x]]$ into $\mathbb{E}[\mathbb{D}[v]]$, provided the context $\mathbb{E}[\mathbb{D}]$ defines x as v and no variable capture occurs. The meta-variable $\mathbb{D}$ ranges over dereferencing contexts. A dereferencing context is a context that expects a non-variable value to fill the hole in order to evaluate. An example of dereferencing context is $\Box\ v$, that is, the function part of a function application. An example of a non-dereferencing context is $(\lambda x.e)\ \Box$, that is, the argument part of a function application, where a variable would allow the evaluation to continue. Dereferencing contexts are formally defined in figure 4. The SUBST rule replaces a variable in a dereferencing context with its value, found in the current top-most binding.

The values of the variables bound by the top-most binding are accessible in two possible ways. If $b_v, x = e, b$ is the partially evaluated, top-most binding, then the already evaluated definitions in b_v can be used for the evaluation of the remaining definitions, beginning with e. Otherwise, if the top-most binding is fully evaluated, then the bound variables can be used to evaluate the enclosed expression. Rules EA and IA in figure 4 capture these two possibilities. They implement the external and internal substitution operations, respectively.

The computational reduction relation on expressions is compatible with structural equivalence. Hence we can define computational reduction over equivalence classes of expressions, obtaining the reduction relation $\longrightarrow$.

Variables	$x \in Vars$			
Names	$X \in Names$			
Locations	$\ell \in Locs$			
Expressions	$E \in Expr$	$::=$	n	integers
		$\mid$	$x \mid \lambda x.E \mid E\ E$	λ-calculus
		$\mid$	let B in E	let
		$\mid$	$\{S\}$	records
		$\mid$	$E.X$	selection
		$\mid$	ℓ	locations
		$\mid$	alloc	allocation
		$\mid$	update	update
Records	S	$::=$	$X_1 = E_1 \ldots X_n = E_n$	
Bindings	B	$::=$	$x_1 = E_1 \ldots x_n = E_n$	

Figure 6: Syntax of λ_{alloc}

Configurations			
	C	$::=$	$\Theta \vdash E$
Heaps			
	$\Theta \in Heaps$	$=$	$Locs \xrightarrow{\text{Fin}} HeapValues$
Answers			
	$A \in Answers$	$::=$	$\Theta \vdash V$
Values			
	$V \in Values$	$::=$	$x \mid \ell \mid n$
Heap values			
	$H_v \in HeapValues$	$::=$	$\lambda x.E \mid \{S_v\} \mid$ alloc n
	S_v	$::=$	$X_1 = V_1 \ldots X_n = V_n$

Figure 7: Configurations and answers in λ_{alloc}

Definition 1 *The $\lambda_\circ$ language is the set of terms equipped with the relation $\longrightarrow$.*

In the remainder of this paper, we study the compilation of the $\lambda_\circ$ language, concentrating on its non-standard let rec construct. Our target language for this compilation is presented in the next section: it is a λ-calculus without a let rec construct, but with support for heap blocks, locations, and in-place update.

4. THE TARGET LANGUAGE

The syntax of the target language λ_{alloc} is presented in figure 6. It includes the λ-calculus with integer constants, and a non-recursive let binding. The expression let $x_1 = E_1 \ldots x_n = E_n$ in E is semantically equivalent to let $x_1 = E_1$ in ... let $x_n = E_n$ in E. Additionally, there are constructs for record operations (creation and selection), and constructs for modeling the heap: an allocation operator alloc, an update operator update, and heap locations ℓ.

The semantics of λ_{alloc} is defined as a reduction relation on configurations. As defined in figure 7, a configuration C is a pair of a heap Θ and an expression E, written $\Theta \vdash E$. A heap is a finite map from locations ℓ to evaluated heap blocks. An evaluated heap block $H_v \in HeapValues$ is either a function $\lambda x.E$, or an evaluated record $\{S_v\}$ (where S_v is an evaluated record sequence of the shape $X_1 = V_1 \ldots X_n = V_n$), or an application of the shape alloc n for some positive integer n. The heap value alloc n represents a dummy heap

block of size n, containing unspecified data. A well-formed configuration is such that all the locations it mentions are bound in its heap.

Evaluated heap blocks are not values. Only integers, variables and locations are values. In this calculus, function abstractions are not values, since their evaluation allocates the function in the heap, and returns its location: the answer of the evaluation of $\lambda x.E$ is a configuration $\Theta \vdash \ell$, where the location ℓ is bound to $\lambda x.E$ in the heap Θ.

The operators related to heaps are alloc, which creates a new empty block of the size given by its argument, and update, which overwrites the contents of its first argument with the contents of its second argument, provided they have the same size. To model this constraint, we assume given a function *Size* from heap values H_v to integers.

Notations We write $\Theta\langle \ell \mapsto H_v \rangle$ for the map equal to Θ anywhere but on ℓ where it returns H_v. We write $\Theta_1 + \Theta_2$ for the union of two heaps Θ_1 and Θ_2 whose domains are disjoint. In particular, when the heap Θ is undefined on ℓ, we write $\Theta + \{\ell \mapsto H_v\}$ to denote the union of Θ and $\{\ell \mapsto H_v\}$.

Structural equivalence and substitutions In λ_{alloc}, expressions are identified up to renaming of bound locations. Locations are bound only by heaps, at top level in configurations. We consider configurations equal modulo renaming of bound locations. This relation is easy to define since the location renamings never cross any location binder.

Moreover, we consider configurations equal modulo renaming of bound variables. However, we will see that the computational reduction relation uses a more complex notion of substitution than just variable renaming: it must also replace variables with locations in some cases. Therefore, we consider variable renaming as a special case of general substitutions, which we now define.

Substitutions are elements of $Subst = Vars \to Values$. The domain of a substitution is the set of variables x such that $\sigma(x) \neq x$. Its codomain is the image of its domain. We write $x\{\sigma\}$ as synonymous for $\sigma(x)$. We often describe substitutions by sets of bindings $\{x_1 \mapsto V_1 \dots x_n \mapsto V_n\}$, implying that their domain is included in the set $\{x_1 \dots x_n\}$. We sometimes consider substitutions as sets, taking the union of two of them when it makes sense, and sometimes we compose them. The composition of σ_1 and σ_2 is defined by $e\{\sigma_2 \circ \sigma_1\} = e\{\sigma_1\}\{\sigma_2\}$: it acts like σ_1 followed by σ_2. Moreover, we call variable renamings, or simply renamings, the injective substitutions whose codomains contain only variables, and we denote them by ζ. Symmetrically, we call variable allocations the injective substitutions mapping variables to locations, and denote them by η.

We extend substitutions to λ_{alloc} expressions and configurations in the usual capture-avoiding manner. A precise definition of substitution is given in the companion technical report [9].

4.1 Semantics

The semantics of λ_{alloc}, like the one of $\lambda_\circ$, is given in terms of a computational contraction relation that handles rules for the basic constructions, and a computational reduction relation that handles global rules. Evaluation answers $\Theta \vdash V$ are values surrounded by a heap binding. (See figure 7.)

Computational contraction relation The computational contraction relation is defined by the rules in figure 8, using the notion of lift contexts.

The Beta rule is unusual in that it applies a heap allocated function to an argument V. The function must be a location ℓ bound in the heap to a value $\lambda x.E$, and the result is $E\{x \mapsto V\}$.

The Project rule works similarly: it projects a name X out of a heap-allocated record $\{S_v\}$ at location ℓ, where S_v is a finite set of evaluated record field definitions of the shape $X_1 = V_1 \dots X_n = V_n$. The result is $S_v(X)$, i.e. V_i if $X = X_i$.

The Allocate rule is one of the key points of λ_{alloc}. It states that a value block H_v evaluates into a fresh heap location containing H_v, and a pointer to it: $\Theta + \{\ell \mapsto H_v\} \vdash \ell$ (ℓ fresh). In particular, if H_v is a dummy block $\mathsf{alloc}\, n$, the result is a dummy block on the heap.

The Update rule copies the contents of a heap block into another heap block. If the locations ℓ_1 and ℓ_2 are respectively bound to blocks H_{v1} and H_{v2} in the heap Θ, then $\Theta \vdash \mathsf{update}\, \ell_1\, \ell_2$ will evaluate to $\Theta\langle \ell_1 \mapsto H_{v2} \rangle \vdash \{\}$.

Finally, as in $\lambda_\circ$, the evaluation of bindings is confined to the top level of configurations. This requires the Lift rule, which lifts a binding outside of a lift context. In λ_{alloc}, lift contexts Λ are defined by

$$\Lambda ::= \square\ E \mid V\ \square \mid \square.X \mid \{\Sigma\} \mid \mathsf{let}\ \ x = \square, B\ \ \mathsf{in}\ \ e,$$

where Σ ranges over record contexts, of the shape $\Sigma ::= S_v, X = \square, S$.

Computational reduction relation The computational reduction relation is defined in figure 8.

The Context rule shifts the contraction relation to a nested lift context Φ. Lift contexts have been defined in the last paragraph, and nested lift contexts are simply series of nested lift contexts.

The Let rule describes the top-level evaluation of bindings. Once the first definition is evaluated, the bound variable is replaced by the obtained value in the rest of the expression. Eventually, the binding becomes empty and can be removed with rule EmptyLet.

By rule GC, when a heap binding is not used by any other binding than itself, and not used either by the expression, it may be removed. (The need for this rule arises from the translation scheme for let rec definitions: after a pre-allocated block has been updated by the contents of the value of the right-hand side expression, the top-most block of this value becomes unreferenced. Rule GC allows to remove this top-most block entirely.)

Finally, the EM rule states that it is equivalent to evaluate two bindings in succession, or to evaluate their union.

4.2 The calculus and its confluence

The computational reduction relation on expressions is compatible with structural equivalence, so we can extend it to terms, obtaining the reduction relation $\longrightarrow$.

Definition 2 *The λ_{alloc} calculus is the set of terms, equipped with the relation $\longrightarrow$.*

Unlike in $\lambda_\circ$, the reduction of λ_{alloc} is not deterministic because of rules GC and EM. Rule GC can apply at any time, and rule EM gives a choice between two possibilities when two successive bindings are encountered. Despite this source of non-determinism, it can be shown that λ_{alloc} is confluent [9].

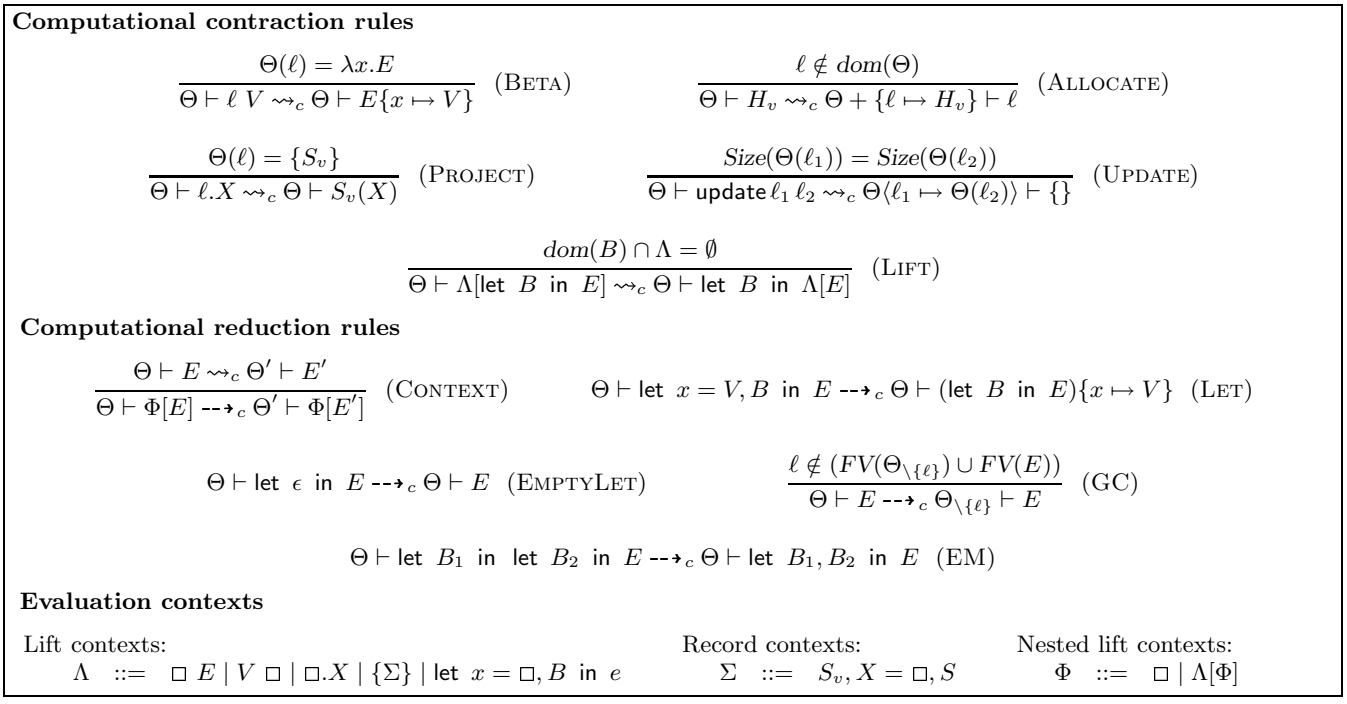


Figure 8: Computational reduction for λ_{alloc}

Theorem 1 (Confluence of λ_{alloc}) *The λ_{alloc} calculus is confluent.*

4.3 Relation to a machine language

While λ_{alloc} is presented above as an extended λ-calculus with reduction semantics, it was carefully engineered to map directly to an abstract machine with a store, and to allow efficient compilation to machine code. In particular, the heaps and locations used in the semantics correspond exactly to machine-level heaps and memory addresses. (This is apparent in the requirement that the update operation works only if the two blocks have the same size.) Actually, the λ_{alloc} calculus is similar to a subset of one of the intermediate languages used by the OCaml compiler, from which it generates efficient native machine code.

5. TRANSLATION

5.1 The standard translation

We now define a translation from $\lambda_\circ$ to λ_{alloc} that implements straightforwardly the in-place update trick. This translation, called the standard translation, is defined in figure 9. It is straightforward for variables, functions, applications, and record operations, but the translation of bindings is more intricate. The translation of a binding b is the concatenation of two bindings in λ_{alloc}. The first binding is called the pre-allocation binding, and gives instructions to allocate dummy blocks on the heap for definitions of known sizes. The second binding is called the update binding. It computes definitions, and either updates the previously preallocated dummy blocks for definitions of known sizes, or simply binds the result for definitions of unknown sizes.

This translation relies on a function *Size* that associates to each $\lambda_\circ$ expression a size indication, which can be either an integer (a number of memory words) or the undefined size, written [?]. This function is supposed to guess the size of the value of the translation of its argument. We assume that the size of any expression of predictable shape is known, and moreover that the size of variables is undefined. In other words, $Size(e_\downarrow) \neq [?]$ for any $e_\downarrow \in Predictable$, and $Size(x) = [?]$ for any variable x,

While perfectly adequate as a compilation scheme in an actual compiler, the standard translation does not lend itself to a correctness proof. Such a correctness proof should be a simulation argument: if $e \longrightarrow e'$ in $\lambda_\circ$, then $[\![e]\!] \longrightarrow^+ [\![e']\!]$; moreover, if e is an answer, $[\![e]\!]$ should be an answer as well. However, both properties fail. For instance, the expression $\lambda x.x$ is an answer in $\lambda_\circ$, but it translates to $\lambda x.x$, which reduces in λ_{alloc} to the configuration $\{\ell \mapsto \lambda x.x\} \vdash \ell$.

Similarly, consider $e = \mathsf{let\ rec}\ y = \lambda x.x\ \mathsf{in}\ f$. If $f \longrightarrow f'$, the expression e reduces to $e' = \mathsf{let\ rec}\ y = \lambda x.x\ \mathsf{in}\ f'$ in $\lambda_\circ$. However, the translations of e and e' are

$$[\![e]\!] = \mathsf{let}\ y = \mathsf{alloc}\,n, y' = \mathsf{update}\,y\ (\lambda x.x)\ \mathsf{in}\ [\![f]\!]$$
$$[\![e']\!] = \mathsf{let}\ y = \mathsf{alloc}\,n, y' = \mathsf{update}\,y\ (\lambda x.x)\ \mathsf{in}\ [\![f']\!]$$

and $[\![e]\!]$ does not reduce to $[\![e']\!]$ in λ_{alloc}: it is not possible to reduce $[\![f]\!]$ until the enclosing let has been reduced.

To overcome this difficulty, we are going to define another translation scheme from $\lambda_\circ$ to λ_{alloc}, called the TOP translation. This alternate translation is less intuitive than the standard translation, but is easier to prove correct using a simulation argument. The correctness of the standard translation follows from that of the TOP translation because the standard translation of a term reduces to its TOP translation.

The intuition behind the TOP translation is that it performs "on the fly" a number of administrative reductions over the result of the standard translation.

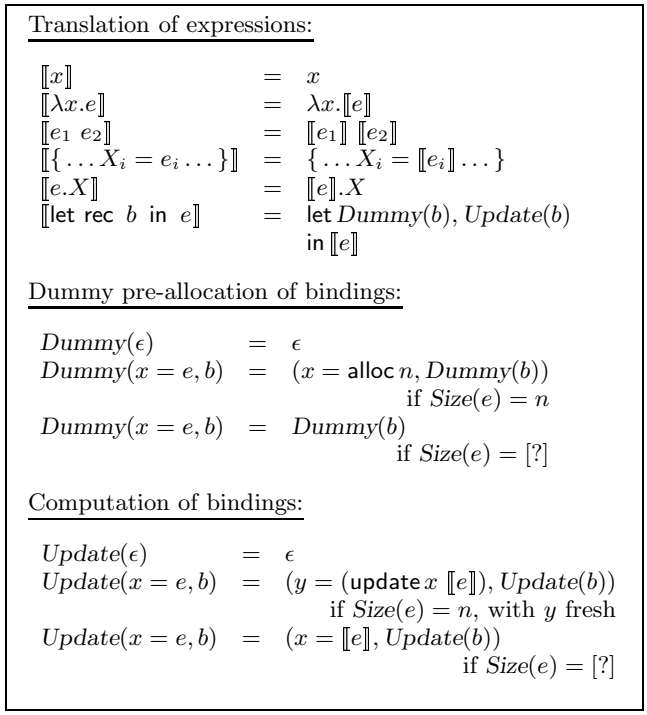


Figure 9: Translation (standard translation)

These additional reductions suffice to ensure, in particular, that answers are mapped to answers. Continuing the example above, the TOP translation maps $\lambda x.x$ to the configuration $\{\ell \mapsto \lambda x.x\} \vdash \ell$, which is an answer.

5.2 Compositionality

As outlined above, the TOP translation maps $\lambda_\circ$ expressions to λ_{alloc} configurations, and not just expressions. An unfortunate consequence of this requirement is that the TOP translation cannot be compositional, in the usual sense: configurations do not compose syntactically. For instance, the translation of an application such as $(\lambda x.x)\ (\lambda x.x)$ is not the application of the translation of the function to the translation of the argument.

To recover some degree of compositionality, we introduce a non-standard notion of contexts in λ_{alloc}, which take as an argument configurations, rather than just expressions. Contexts are pairs of a heap and a nested lift context, and the application of a context $\Theta \vdash \Phi$ to a configuration $\Theta' \vdash E$ is the configuration $\Theta + \Theta' \vdash \Phi[E]$.

This is not sufficient, however. Recall that answers in $\lambda_\circ$ can be of the shape $b_v \vdash v$. Intuitively, b_v should be translated as a heap. But heaps of λ_{alloc} only contain heap blocks, i.e. dummy blocks, functions or evaluated records, while the binding b_v can also contain definitions of the shape $x = y$ for example (or $x = 1$ if $\lambda_\circ$ featured constants), which we do not want to translate as heap bindings. Furthermore, we have to take into account the asymmetry of $\mathsf{let\ rec}$ in $\lambda_\circ$. Indeed, the heap $x = y, z = x$ maps both x and z to the value y. Our solution is to retain the part of $\lambda_\circ$ heaps that cannot be included in λ_{alloc} heaps as substitutions. For instance, the $\lambda_\circ$ binding $x = y, z = x$ is translated as the substitution $\{x \mapsto y\} \circ \{z \mapsto x\}$. This approach complicates the notion of contexts: now, they must include a substitution. Indeed, the $\lambda_\circ$ context $x = y, z = x \vdash \square$ does not correspond to any standard evaluation context in λ_{alloc}. Instead, we have to define a stronger kind of evaluation contexts, including a heap Θ, a standard context Φ, and a substitution σ. We write these extended contexts $\Theta \vdash \Phi[\sigma]$, and denote them by Ψ.

Applying a context to a configuration is valid if the two heaps define disjoint sets of locations, and if the substitution carried by the context is correct for the configuration. Fortunately, when the proposed substitution is not correct for the considered configuration, structural equivalence allows to rename all the problematic binders in it, and find an equivalent configuration for which the substitution is correct. The application of a context $\Theta \vdash \Phi[\sigma]$ to a configuration $\Theta' \vdash E$ is the configuration $(\Theta + \Theta' \vdash \Phi[E])\{\sigma\}$.

Similarly, the composition $\Psi_1 \circ \Psi_2$ of two contexts $\Psi_i = \Theta_i \vdash \Phi_i[\sigma_i]$ is $\Theta_1 + \Theta_2 \vdash \Phi_1[\Phi_2][\sigma_1 \circ \sigma_2]$, provided the substitution $\sigma_1 \circ \sigma_2$ is correct for the heap $\Theta_1 + \Theta_2$ and the context $\Phi_1[\Phi_2]$. Fortunately, in λ_{alloc} contexts, binders are not in position to capture the placeholder, so structural equivalence always allows to find correct, equivalent contexts.

5.3 Definition of the TOP translation

TOP translation of expressions The TOP translation, defined in figures 10 and 11, associates λ_{alloc} configurations to $\lambda_\circ$ expressions, and λ_{alloc} configurations to $\lambda_\circ$ configurations.

The idea is that the TOP translation is used until the current point of evaluation in the expression, and beyond that point, the standard translation is used.

Variables are still translated as variables. A function $\lambda x.e$ is translated as with the standard translation, i.e. $\lambda x.[\![e]\!]$, but the result is allocated on the heap, at a fresh location ℓ: $\{\ell \mapsto \lambda x.[\![e]\!]\} \vdash \ell$.

The translation of an evaluated record takes the translations of its fields and puts them in a record allocated on the heap at a fresh location ℓ, obtaining $\Theta + \{\ell \mapsto \{S_v\}\} \vdash \ell$. Here, $\Theta \vdash S_v$ is the translation of the record s_v, defined in figure 10. If $s_v = (X_1 = v_1 \dots X_n = v_n)$, and for each i, $[\![v_i]\!]^{\mathrm{TOP}} = \Theta_i \vdash V_i$, then $\Theta \vdash S_v = \biguplus_{1 \le i \le n} \Theta_i \vdash (X_1 = V_1 \dots X_n = V_n)$.

When the record is not fully evaluated, it is not yet allocated on the heap. It is divided into its evaluated part s_v, and the rest $X = e, s$. The s_v part is translated as for evaluated records, into $\Theta_1 \vdash S_v$. The field e is translated with the TOP translation, into $\Theta_2 \vdash E$, and s is translated with the standard translation. We denote by $[\![s]\!]$ the record s, translated with the standard translation. The result is $\Theta_1 + \Theta_2 \vdash \{S_v, X = E, [\![s]\!]\}$.

Function application works like records: if the function part is not a value, then it is translated with the TOP translation, while the argument is translated with the standard translation. If the function is a value, then both parts are translated with the TOP translation.

The translation of a record selection $e.X$ consists of translating e with the TOP translation, and then selecting the field X.

TOP translation of bindings The TOP translation of bindings is more complicated. As for records, the binding is divided into its evaluated part b_v and the rest b, which can be empty, but does not begin with a value.

The unevaluated part of the binding, b, is translated as

Translation of expressions into configurations:

$$
\begin{array}{lcll}
[\![x]\!]^{\mathrm{TOP}} & = & \emptyset \vdash x & \\
[\![\lambda x.e]\!]^{\mathrm{TOP}} & = & \{\ell \mapsto \lambda x.[\![e]\!]\} \vdash \ell & \\
[\![\{s_v\}]\!]^{\mathrm{TOP}} & = & \Theta + \{\ell \mapsto \{S_v\}\} \vdash \ell & \text{for } [\![s_v]\!]^{\mathrm{TOP}} = \Theta \vdash S_v \\
[\![\{s_v, X = e, s\}]\!]^{\mathrm{TOP}} & = & \Theta_1 + \Theta_2 \vdash \{S_v, X = E, [\![s]\!]\} & \text{for } \left\{ \begin{array}{l} e \notin \mathit{values} \\ [\![s_v]\!]^{\mathrm{TOP}} = \Theta_1 \vdash S_v \\ [\![e]\!]^{\mathrm{TOP}} = \Theta_2 \vdash E \end{array} \right. \\
[\![v\ e]\!]^{\mathrm{TOP}} & = & \Theta_1 + \Theta_2 \vdash VE & \text{for } \left\{ \begin{array}{l} [\![v]\!]^{\mathrm{TOP}} = \Theta_1 \vdash V \\ [\![e]\!]^{\mathrm{TOP}} = \Theta_2 \vdash E \end{array} \right. \\
[\![e_1\ e_2]\!]^{\mathrm{TOP}} & = & \Theta \vdash E[\![e_2]\!] & \text{for } \left\{ \begin{array}{l} e_1 \notin \mathit{values} \\ [\![e_1]\!]^{\mathrm{TOP}} = \Theta \vdash E \end{array} \right. \\
[\![e.X]\!]^{\mathrm{TOP}} & = & \Theta \vdash E.X & \text{for } [\![e]\!]^{\mathrm{TOP}} = \Theta \vdash E \\
[\![\mathsf{let\ rec}\ b\ \mathsf{in}\ e]\!]^{\mathrm{TOP}} & = & \left\{ \begin{array}{ll} [\![b]\!]^{\mathrm{TOP}}[\emptyset \vdash [\![e]\!]] & \text{if } b \text{ is not evaluated} \\ [\![b]\!]^{\mathrm{TOP}}[[\![e]\!]^{\mathrm{TOP}}] & \text{otherwise} \end{array} \right. &
\end{array}
$$

Translation of configurations:

$$[\![b \vdash e]\!]^{\mathrm{TOP}} = [\![\mathsf{let\ rec}\ b\ \mathsf{in}\ e]\!]^{\mathrm{TOP}}$$

Translation of bindings and evaluated records:

$$
\begin{array}{lcll}
[\![b_v, b]\!]^{\mathrm{TOP}} & = & \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b) & \text{where } b \neq (x = v, b') \\
[\![X_1 = v_1 \ldots X_n = v_n]\!]^{\mathrm{TOP}} & = & \biguplus_{1 \leq i \leq n} \Theta_i \vdash (X_1 = V_1 \ldots X_n = V_n) & \text{with } \forall i, [\![v_i]\!]^{\mathrm{TOP}} = \Theta_i \vdash V_i
\end{array}
$$

Figure 10: The TOP translation (first part)

Translation of evaluated bindings: Evaluated binding → (heap × substitution × variable allocation)

$$
\begin{array}{lcll}
\mathit{TOP}(\epsilon) & = & \emptyset \vdash (id, id) & \\
\mathit{TOP}(x = y, b_v) & = & \Theta \vdash (\{x \mapsto y\} \circ \sigma, \eta) & \text{if } \mathit{TOP}(b_v) = \Theta \vdash (\sigma, \eta) \\
\mathit{TOP}(x = v, b_v) & = & \Theta_1 + \Theta_2 \vdash (\sigma, \eta \cup \{x \mapsto \ell\}) & \text{if } \left\{ \begin{array}{l} v \notin \mathit{Vars} \\ [\![v]\!]^{\mathrm{TOP}} = \Theta_1 \vdash \ell \\ \mathit{TOP}(b_v) = \Theta_2 \vdash (\sigma, \eta) \end{array} \right.
\end{array}
$$

Actual dummy pre-allocation: Binding → (heap × variable allocation)

$$
\begin{array}{lcll}
\mathit{TDum}(\epsilon) & = & \emptyset \vdash id & \\
\mathit{TDum}(x = e, b) & = & \mathit{TDum}(b) & \text{if } \mathit{Size}(e) = [?] \\
\mathit{TDum}(x = e, b) & = & \Theta + \{\ell \mapsto \mathsf{alloc}\, n\} \vdash \eta \cup \{x \mapsto \ell\} & \text{if } \left\{ \begin{array}{l} \mathit{Size}(e) = n \\ \mathit{TDum}(b) = \Theta \vdash \eta \end{array} \right.
\end{array}
$$

Actual computation of bindings: Binding of $\lambda_\circ$ → (heap × binding of λ_{alloc})

$$
\begin{array}{lcll}
\mathit{TUp}(\epsilon) & = & \emptyset \vdash \epsilon & \\
\mathit{TUp}(x = e, b) & = & \Theta_1 + \Theta_2 \vdash x = E, B & \text{if } \left\{ \begin{array}{l} \mathit{Size}(e) = [?] \\ [\![e]\!]^{\mathrm{TOP}} = \Theta_1 \vdash E \\ \mathit{TUp}(b) = \Theta_2 \vdash B \end{array} \right. \\
\mathit{TUp}(x = e, b) & = & \Theta_1 + \Theta_2 \vdash y = (\mathsf{update}\, x\ E), B & \text{if } \left\{ \begin{array}{l} \mathit{Size}(e) \neq [?] \\ [\![e]\!]^{\mathrm{TOP}} = \Theta_1 \vdash E \\ \mathit{TUp}(b) = \Theta_2 \vdash B \\ y \text{ fresh} \end{array} \right.
\end{array}
$$

Figure 11: The TOP translation (continued): bindings

follows. In the standard translation, the pre-allocation pass, consists in giving instructions for allocating dummy blocks. Here, these blocks are directly allocated by the function *TDum*, which returns the heap of dummy blocks, and the substitution replacing variables with the corresponding locations. The update pass, in the standard translation, either updates a dummy block with the translation of the definition, or simply binds it. In the TOP translation, the only difference is that the first definition is translated with the TOP translation, while the remaining ones are translated with the standard translation. The function *TUp* is in charge of these operations.

The evaluated part of the binding, b_v, is translated as a heap and a substitution, by the *TOP* function. A definition of unknown size $x = v$ yields a translation of the shape $\emptyset \vdash V$, and is included in the translation as the substitution $x \mapsto V$. A definition of known size $x = v$ is translated as a heap and a variable allocation: v has a translation of the shape $\Theta \vdash \ell$, and it is included in the translation of b_v as Θ, and the allocation $x \mapsto \ell$.

In practice, it is useful to distinguish substitutions coming from definitions of unknown sizes, which can be of any shape, from substitutions coming from definitions of known sizes, which are allocations, and therefore have the shape $x \mapsto \ell$. Indeed, when putting the results together, it is important to take the order into account for definitions of unknown sizes. For instance, as noticed above, a binding such as $x = y, z = x$ generates two substitutions $x \mapsto y$ and $z \mapsto x$, but the former must be performed last. This is why, according to the definition of *TOP*, the result is $\{x \mapsto y\} \circ \{z \mapsto x\}$. This works because, due to the syntactic restrictions on let rec, definitions of unknown sizes can only be mentioned by subsequent definitions in the binding. However, definitions of known sizes can be mentioned by previous definitions. The key observation is that the substitutions they generate are allocations, so they are not modified by other substitutions, and can be performed last. Formally, the translation of b_v is a heap Θ, a substitution σ, corresponding to the definitions of unknown sizes, and an allocation η, giving the locations allocated in Θ for the definitions of known sizes. Semantically, it corresponds to a heap Θ and the substitution $\eta \circ \sigma$, and will be used as such.

The three functions for translating bindings, *TDum*, *TUp*, and *TOP*, can be viewed as contexts. The *TDum* function returns a heap Θ and an allocation η, which form a context $\Theta \vdash \Box[\eta]$. The *TUp* function returns a heap Θ and a binding B, which form a context $\Theta \vdash \mathsf{let}\ B\ \mathsf{in}\ \Box[id]$. The *TOP* function returns a heap Θ, a substitution σ, and an allocation η, which form a context $\Theta \vdash \Box[\eta \circ \sigma]$. Notice that the context corresponding to *TUp* is not an evaluation context. Fortunately, the substitutions that are applied to it do not involve the domain of its binding, thus preserving the meaning. In case the whole binding b_v, b is evaluated (i.e. b is empty), then the contexts for pre-allocation and update, $TDum(b)$ and $TUp(b)$ are empty, and the translation of $\mathsf{let\ rec}\ b_v, b\ \mathsf{in}\ e$ is the TOP translation of e put in the context $TOP(b_v)$. Otherwise, the translation of $\mathsf{let\ rec}\ b_v, b\ \mathsf{in}\ e$ is the standard translation of e, put in the context $TDum(b) \circ TOP(b_v) \circ TUp(b)$.

5.4 Relating the two translations

An interesting fact is that the standard translation of any expression reduces to its TOP translation, in any context, provided the following hypotheses on the *Size* function are met.

Hypothesis 1 *For all expressions e, f, e', value v, bindings b, b', substitution σ, and context $\mathbb{C}$:*

- *If $Size(e) = n$ and $b \vdash e \longrightarrow b' \vdash e'$, then $Size(e') = n$.*
- *If $Size(v) = n$, then there exist Θ and ℓ such that $[\![v]\!]^{\mathrm{TOP}} = \Theta \vdash \ell$ and $Size(\Theta(\ell)) = n$.*
- *$Size(e) = Size(f) = n$ implies $Size(\mathbb{C}[e]) = Size(\mathbb{C}[f])$.*
- *$Size(e\{\sigma\}) = Size(e)$.*
- *$Size(\mathsf{let\ rec}\ b\ \mathsf{in}\ e) = Size(e)$.*

Lemma 1 *For all contexts Ψ and for all expressions e,*

$$\Psi[\emptyset \vdash [\![e]\!]] \longrightarrow^* \Psi[[\![e]\!]^{\mathrm{TOP}}].$$

6. CORRECTNESS OF THE TRANSLATION

Owing to their different ways of handling bindings, the two languages $\lambda_\circ$ and λ_{alloc} do not yield a step-by-step simulation. Indeed, a redex and its reduct in $\lambda_\circ$ may have the same translation. As an example, consider two expressions of the shape $\mathbb{L}[\mathsf{let\ rec}\ b_v\ \mathsf{in}\ e]$ and $\mathsf{let\ rec}\ b_v\ \mathsf{in}\ \mathbb{L}[e]$. The binding b_v is translated as a heap Θ and a substitution σ, in both cases, and the fact that it is under or above the $\mathbb{L}$ context is not visible in the translation. This gives rise to a "stuttering problem": conceivably, an infinite reduction sequence in $\lambda_\circ$ could be translated to no reduction at all in λ_{alloc}, thus changing the termination behavior of the program. In order to ensure that this cannot happen, we prove that such silent reduction steps cannot happen indefinitely. For this, we introduce a measure μ on expressions and configurations that strictly decreases during silent reduction steps. Its precise definition is given in the companion technical report [9]. Intuitively, the three kinds of silent steps cause a decrease in a syntactic feature of the term:

- internal or external merge steps strictly decrease the number of let rec nodes;
- lift steps move a let rec node up one level toward the top;
- internal or external substitution steps replace a variable with another variable bound earlier in the expression.

The last obstacle to the simulation theorem is the different sharing properties of the two languages. Consider the configuration $c = (x = \{X = \lambda y.y\} \vdash (x.X)\ x)$. It reduces by rule Subst to $c' = (x = \{X = \lambda y.y\} \vdash (\{X = \lambda y.y\}.X)\ x)$. By the TOP translation, c is translated to a configuration

$$C = \left\{ \begin{array}{l} \ell_1 \mapsto \lambda y.y, \\ \ell_2 \mapsto \{X = \ell_1\} \end{array} \right\} \vdash (\ell_2.X)\ \ell_2.$$

By the same translation, c' is translated to a configuration

$$C' = \left\{ \begin{array}{l} \ell_1 \mapsto \lambda y.y, \\ \ell_2 \mapsto \{X = \ell_1\}, \\ \ell_3 \mapsto \lambda y.y, \\ \ell_4 \mapsto \{X = \ell_3\} \end{array} \right\} \vdash (\ell_4.X)\ \ell_2.$$

The heap Θ' of C' contains an additional copy of the record and the function. This phenomenon happens at each application of the SUBST rule. But, except in case of a faulty configuration, such a reduction step can be followed immediately by a BETA or a PROJECT step. In our example, a PROJECT step occurs in $\lambda_\circ$: c' reduces to $c'' = (x = \{X = \lambda y.y\} \vdash (\lambda y.y)\ x)$. This reduction step destroys the copied record immediately after it has been copied. Similarly, when a function is copied, the copy is immediately destroyed by the subsequent BETA reduction step. In both cases, the translated configuration reduces in one step, by the same rule (PROJECT or BETA). As a consequence, our simulation theorem takes this possibility into account, and allows a couple of successive reduction steps to be simulated by a single one.

However, in the case of the PROJECT rule, not only the record is duplicated, but also the values it contains. In our example, the function $\lambda y.y$ is copied. And even after applying the PROJECT rule, it remains, as shown by the translation of c'':

$$C'' = \left\{ \begin{array}{l} \ell_1 \mapsto \lambda y.y, \\ \ell_2 \mapsto \{X = \ell_1\}, \\ \ell_3 \mapsto \lambda y.y \end{array} \right\} \vdash \ell_3\ \ell_2.$$

Our solution to this problem consists in considering only expressions where all the record fields are variables, which we call **R**-normal expressions. Any expression can be transformed into an **R**-normal one, by applying the following NAMEFIELDS rule, in any context.

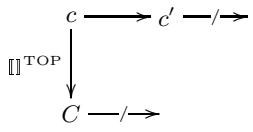
$$\frac{\exists i, e_i \notin \mathit{Vars} \qquad \forall i, j, x_i \notin \mathit{FV}(e_j)}{\begin{array}{l}\{X_1 = e_1 \ldots X_n = e_n\} \xrightarrow{\mathbf{R}} \\ \quad \mathsf{let\ rec}\ x_1 = e_1 \ldots x_n = e_n \\ \quad \mathsf{in}\ \{X_1 = x_1 \ldots X_n = x_n\}\end{array}} \quad (\textsc{NameFields})$$

This process necessarily terminates since the number of records containing expressions other than variables strictly decreases. The reduction rules of $\lambda_\circ$ obviously preserve **R**-normality. This way, after a sequence of a SUBST step followed by a PROJECT step, no duplication has been made: an expression of the shape $x.X$ has been replaced with another variable.

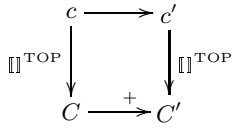
We can now state our main theorem. A $\lambda_\circ$ configuration is said to be stuck on a free variable when it is of the shape $\mathbb{E}\,[\mathbb{D}\,[x]]$ and $\mathbb{E}\,(x)$ is undefined. This definition is extended to λ_{alloc} configurations (replace $\mathbb{E}$ with Ψ and $\mathbb{D}$ with the obvious notion of dereferencing contexts for λ_{alloc}). We say that a configuration is faulty if it is in normal form and is not a valid answer and is not stuck on a free variable.

Theorem 2 (Simulation) *For all **R**-normal configuration c, if $c \longrightarrow c'$ and $[\![c]\!]^{\mathrm{TOP}} = C$, then one of the four situations below holds.*

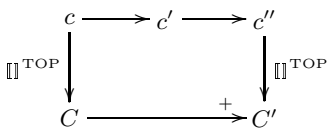
1. *Either c' and C are faulty.*



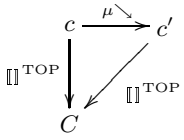
2. *Or, there exists C' such that $[\![c']\!]^{\mathrm{TOP}} = C'$ and $C \longrightarrow^+ C'$.*



3. *Or there exists c'', C' such that $[\![c'']\!]^{\mathrm{TOP}} = C'$, $c \longrightarrow c''$, and $C \longrightarrow^+ C'$.*



4. *Or $[\![c']\!]^{\mathrm{TOP}} = C$ directly, and $\mu(c) > \mu(c')$.*



As a corollary, we obtain the correctness of the translation.

Theorem 3 (Correctness) *For all expression e in **R**-normal form:*

1. *If $\emptyset \vdash e \longrightarrow^* a$, then $\emptyset \vdash [\![e]\!] \longrightarrow^* [\![a]\!]^{\mathrm{TOP}}$.*
2. *If e goes wrong, i.e. $\emptyset \vdash e$ reduces to a faulty configuration, then $[\![e]\!]$ also goes wrong.*
3. *If e loops, i.e. there exists an infinite reduction sequence starting from $\emptyset \vdash e$, then $[\![e]\!]$ also loops.*
4. *If e gets stuck on a free variable, then so does $[\![e]\!]$.*

While our initial goal was to prove the correctness of our compilation scheme, a completeness result also follows from theorem 2.

Theorem 4 (Completeness) *For all expression e in **R**-normal form:*

1. *If $\emptyset \vdash [\![e]\!] \longrightarrow^* A$, then there exists a such that $\emptyset \vdash e \longrightarrow^* a$ and $[\![a]\!]^{\mathrm{TOP}} = A$.*
2. *If $[\![e]\!]$ goes wrong, then e also goes wrong.*
3. *If $[\![e]\!]$ loops, then e also loops.*
4. *If $[\![e]\!]$ gets stuck on a free variable, then so does e.*

Remark 1 (Free variables) *Free variables do not appear during reduction. Thus, evaluation never gets stuck on a free variable if the initial expression is closed.*

7. RELATED WORK

Cyclic explicit substitutions Rose [14] defines a calculus with mutually recursive definitions, where recursion is introduced by explicit cyclic substitutions, extending the explicit substitutions of Abadi et al. [1]. Instead of lifting recursive bindings to the top of terms like we do, Rose's calculus pushes them inside terms, as usual with explicit substitutions. This results in the loss of sharing information. Any term is allowed in recursive bindings, but inside a recursive binding, when computing a definition, it is not possible to use the value of any definition from the same binding. In $\lambda_\circ$, the rule for substitution SUBST allows this, in conjunction with the internal access rule IA. In Rose's calculus, correct call-by-value reduction requires that in any binding, recursive definitions reduce to values, without using the value of each other. In this respect, it is less powerful than $\lambda_\circ$. Besides, it does not impose size constraints on

definitions, but does not address the issue of efficient data representation.

Benaissa et al. [3] study sharing and different evaluation strategies, for a slightly different notion of cyclic explicit substitution. Any term is accepted in a recursive definition, but instead of going wrong when the recursive value is really needed, as in our system, the system of Benaissa et al. loops. The focus of the paper is on the comparison between λ-graph reduction and environment based evaluation, and different evaluation strategies. No emphasis is put on data representation either.

Equational theories of the λ-calculus with explicit recursion Ariola et al. [2] study a λ-calculus with explicit recursion. Its semantics is given by source-to-source rewrite rules, where let rec is lifted to the top of terms, and definitions in a binding may use each other, as in $\lambda_\circ$. The semantics of our source language $\lambda_\circ$ is largely inspired by their call-by-value calculus. Thus, our work can be seen as transferring the internal substitution rule IA from equational theory to actual language design. Nevertheless, the concerns are different: we deal with implementation and data representation, while Ariola et al. focus on confluence, sharing and different evaluation strategies, including strong reduction (reduction under λ-abstraction).

let rec for objects and mixin modules The let rec constructs used by Boudol [4] and Hirschowitz and Leroy [8] differ from the one of $\lambda_\circ$ in several aspects. First, they accept strictly more expressions as recursive definitions. For instance, Boudol's semantics of objects makes extensive use of recursive definitions such as let rec $o = \mathit{generator}(o)$ in e. Such definitions are not allowed in $\lambda_\circ$. However, $\lambda_\circ$ allows to define in the same binding recursive values and computations using these values. The semantics of mixin modules [10] requires complex sequences of alternate recursive and non-recursive bindings, which are trivial to write in $\lambda_\circ$. Moreover, compared to Boudol's language, the restrictions of $\lambda_\circ$ allow for more efficient execution, since additional indirections are avoided.

8. CONCLUSION AND FUTURE WORK

We have presented and proved correct an efficient compilation scheme for call-by-value evaluation of mutually recursive definitions. The recursive definitions supported by this scheme go beyond recursive functions, and include recursive data structures, as well as the interleaving of recursive and non-recursive bindings in a single let rec construct. These results are relevant to the efficient implementation of call-by-value mixin modules. Additionally, they formally justify the compilation scheme for non-functional let rec definitions used in the OCaml compiler.

In future work, we plan to extend further the class of let rec definitions supported by the compilation scheme. Consider a language where the right-hand sides of recursive definitions are arbitrary expressions, optionally annotated with integers representing the expected sizes for the r.h.s. expressions. This language can be compiled exactly like $\lambda_\circ$: r.h.s. expressions annotated with sizes are treated as having predictable shape, with pre-allocation and in-place update, while unannotated r.h.s. expressions are handled by evaluation and binding. This language is more expressive than $\lambda_\circ$, since it can evaluate recursive definitions such as $o = \mathit{generator}(o)$ provided the size of the result of *generator* can be predicted. For some typed compilation schemes, the size of the result of an expression is a function of the static type of the expression, and can easily be predicted. In other settings, compile-time determination of sizes can be achieved by static analysis.

9. REFERENCES

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *J. Func. Progr.*, 1(4):375–416, 1991.

[2] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1–3):95–178, 2002.

[3] Z.-E.-A. Benaissa, P. Lescanne, and K. H. Rose. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Prog. Lang., Impl., Logics, and Programs*, volume 1140 of *LNCS*, pages 393–407, 1996.

[4] G. Boudol. The recursive record semantics of objects revisited. In D. Sands, editor, *Europ. Symp. on Progr.*, volume 2028 of *LNCS*, pages 269–283. Springer-Verlag, 2001.

[5] G. Boudol and P. Zimmer. Recursion in the call-by-value lambda-calculus. Fixed Points in Comp. Sc. 2002.

[6] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[7] The Haskell language. `http://www.haskell.org`.

[8] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *Europ. Symp. on Progr.*, volume 2305 of *LNCS*, pages 6–20, 2002.

[9] T. Hirschowitz, X. Leroy, and J. B. Wells. On the implementation of recursion in call-by-value functional languages. Research report RR-4728, INRIA, February 2003.

[10] T. Hirschowitz, X. Leroy, and J. B. Wells. A reduction semantics for call-by-value mixin modules. Research report RR-4682, INRIA, January 2003.

[11] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml 3.06 reference manual*, 2002. Available at `http://caml.inria.fr/`.

[12] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2003.

[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (revised)*. The MIT Press, 1997.

[14] K. H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Rémy, editors, *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, volume 656 of *LNCS*, pages 36–50. Springer-Verlag, 1992.

[15] O. Waddell, D. Sarkar, and R. K. Dybvig. Robust and effective transformation of letrec. In *Electronic proceedings of the 2002 Scheme Workshop*, 2002. `http://scheme2002.ccs.neu.edu/`.