On the implementation of recursion in call-by-value functional languages

Tom Hirschowitz, Xavier Leroy, J. B. Wells

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the implementation of recursion in call-by-value functional languages

Tom Hirschowitz — Xavier Leroy — J. B. Wells

N° 4728

Février 2003

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4728--FR+ENG

INRIA
ROCQUENCOURT

On the implementation of recursion in call-by-value functional languages

Tom Hirschowitz * , Xavier Leroy * , J. B. Wells †

Thème 2 — Génie logiciel
et calcul symbolique
Projet Cristal

Rapport de recherche n° 4728 — Février 2003 — 38 pages

Abstract: Functional languages encourage the extensive use of recursive fonctions and data structures. It is therefore important that they efficiently implement recursion. In this paper, we formalize and improve a known implementation technique for recursion. The original technique was introduced by Cousineau and Mauny as the "in-place updating trick". Consider a list of mutually recursive definitions. The technique consists in allocating a dummy, uninitialized heap block for each recursive definition. The size of these blocks is guessed from the shape of each definition. Then, the right-hand sides of the definitions are computed. Recursively-defined identifiers thus refer to the corresponding dummy blocks. This leads, for each definition, to a block of the expected size. Eventually, the contents of the obtained blocks are copied to the dummy blocks, updating them in place. The only change we propose to this scheme is to update the dummy blocks as soon as possible, immediately after each definition is computed, thus making it available for further use. At the source language level, the improvement allows to extend the class of expressions allowed as right-hand sides of recursive definitions, usually restricted to syntactic functions. We formalize our technique as a translation scheme to a lambda-calculus featuring in-place updating of memory blocks, and prove the translation to be faithful.

Key-words: recursion, compilation, semantics.

* Inria Rocquencourt, projet Cristal
† Heriot-Watt University, Edinburgh, UK

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Implémentation de la récursion dans les langages fonctionnels en appel par valeur

Résumé : Les langages fonctionnels encouragent l'utilisation extensive des fonctions et structures de données récursives. Ils doivent donc implanter efficacement la récursion. Dans ce papier, nous en formalisons et améliorons une technique connue d'implantation. Cette technique a été introduite par Cousineau et Mauny sous le nom de "in-place updating trick", soit "l'astuce de la modification en place". Considérons une liste de définitions récursives. La technique consiste à allouer sur le tas des blocs non initialisés, dont les tailles sont devinées à partir de la forme syntaxique des définitions. Les définitions sont ensuite calculées. Les variables récursives font alors référence aux blocs non initialisés. On obtient, pour chaque définition, un bloc de la taille attendue. Enfin, ces blocs obtenus sont copiés sur les blocs non initialisés, ce qui les modifie en place. Le seul changement que nous proposons est de modifier les blocs en place plus tôt, précisément dès qu'il est possible de le faire. Ainsi, dès qu'une définition est calculée, on copie le résultat sur le bloc non initialisé correspondant, ce qui rend la définition disponible lors du calcul des définitions suivantes. L'amélioration apportée permet, au niveau du langage source, d'élargir la classe d'expressions acceptées comme membres droits des définitions récursives, habituellement limitée aux seules fonctions syntaxiques. Nous formalisons notre technique comme un encodage vers un lambda-calcul doté d'opérations pour la modification en place, et nous prouvons que l'encodage est correct.

Mots-clés : récursion, compilation, sémantique.

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 [12] only allows syntactic λ-abstractions, and OCaml [11, 10] allows both λ-abstractions and limited forms of constructor applications.

Several criterion come into play when determining this range. 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 by the system, this code is executed, but it begins by requesting its own execution. 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, 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 criterions 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 [4], right-hand sides of recursive definitions are not syntactically restricted, but ill-founded definitions are ruled out by a type system. More recent work by Hirschowitz and Leroy [8] led to a very similar extension. 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 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. Finally, the compilation scheme we propose for this flavor of **let rec** is simpler and more efficient than that of [5], since it does not require additional reference cells.

Our main motivation in studying this extended **let rec** construct is that it plays a crucial role in the language of call-by-value mixin modules investigated by the authors in [9]. Moreover, the present paper proves the correctness of the folklore "in-place updating trick" [6] used in the OCaml compiler.

The remainder of this paper is organized as follows. In section 2, we first review informally the "in-place updating trick" described in [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 updating of memory blocks. We define the compilation scheme from $\lambda_\circ$ to λ_{alloc} in section 5, and prove its correctness in section 6.

2 Overview

The "in-place updating trick" The "in-place updating trick" outlined in [6] and refined in the OCaml compiler [10], implements **let rec** definitions that satisfy the following two conditions. Consider the 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, a recursive definition like `f = λ x.(... f ...)` is accepted, since no computation will try to use the value of f. Contrarily, a recursive definition like `f = (f 0)` is refused.

Evaluation of a **let rec** definition with in-place updating 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, in a given language L, 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

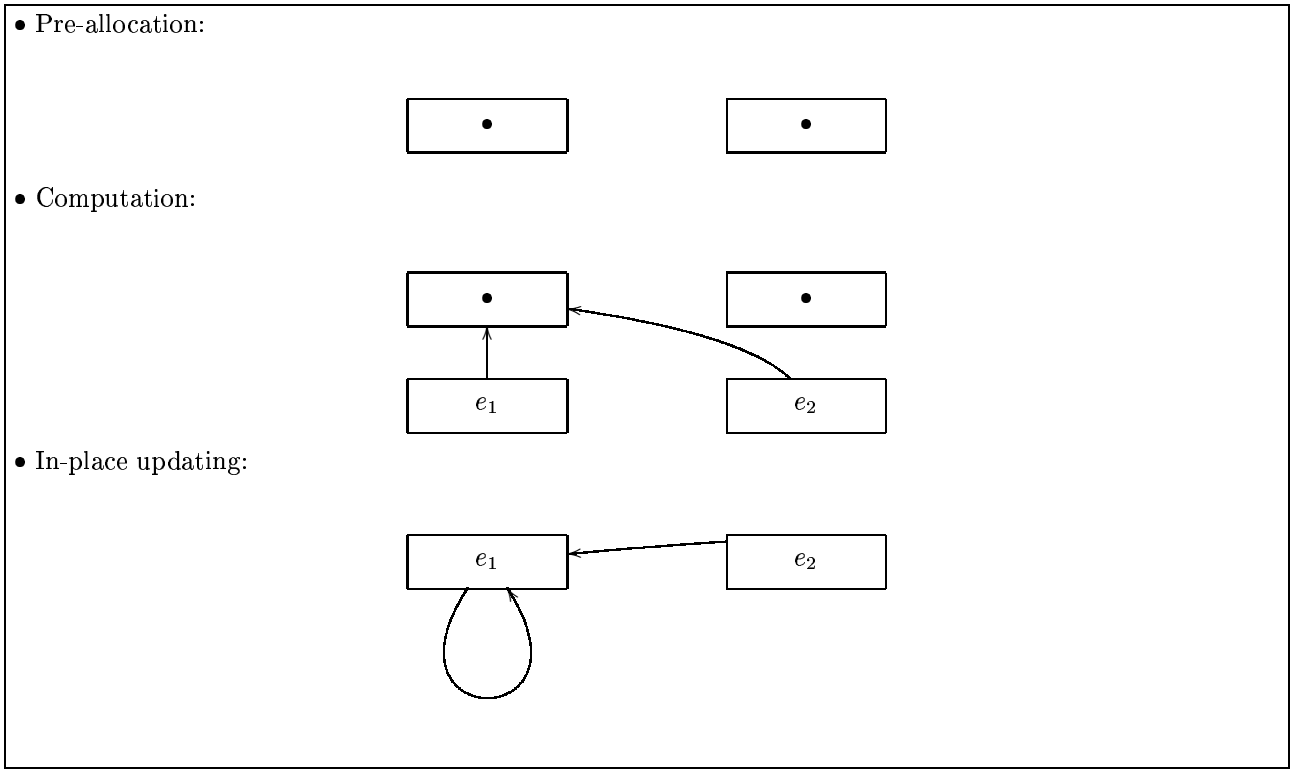


Figure 1: The "in-place updating trick"

allocated blocks of sizes n_1 and n_2, respectively. Here is what the compiled code does, as depicted in figure 1. First, it allocates two uninitialized heap blocks, at adresses l_1 and l_2, of sizes n_1 and n_2, respectively. 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 allocated block of size n_1, with possible references to the two uninitialized blocks. The same process is carried on for e_2, resulting in a heap allocated block of size n_2. The third and final step consists in copying 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 structure.

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

As long as mutual references do not really use the referenced values, as happens for recursive functions for instance, both schemes behave identically. Nevertheless, in the case where e_2 really uses the value v_1 computed for e_1, for example if $e_2 = (x_1\ 1)$, the original scheme can go wrong. Indeed, the dummy block pre-allocated for x_1 is still empty at the time where e_2 is computed. Instead, with immediate in-place updating, the value v_1 is already available when computing e_2. This trivial modification to the scheme thus corresponds to increasing the expressive power of **let rec**. It allows definitions to really use previous definitions. Furthermore, it allows to transparently introduce definitions with unknown sizes in **let rec**, as shown by the following example.

An example of execution is presented in figure 2. The executed 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 n_1 and n_3, 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 make references to x_1 and x_3, which correspond to pointers to the dummy blocks, but not to 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. It can refer to both dummy blocks, but it can also really use the value v_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.

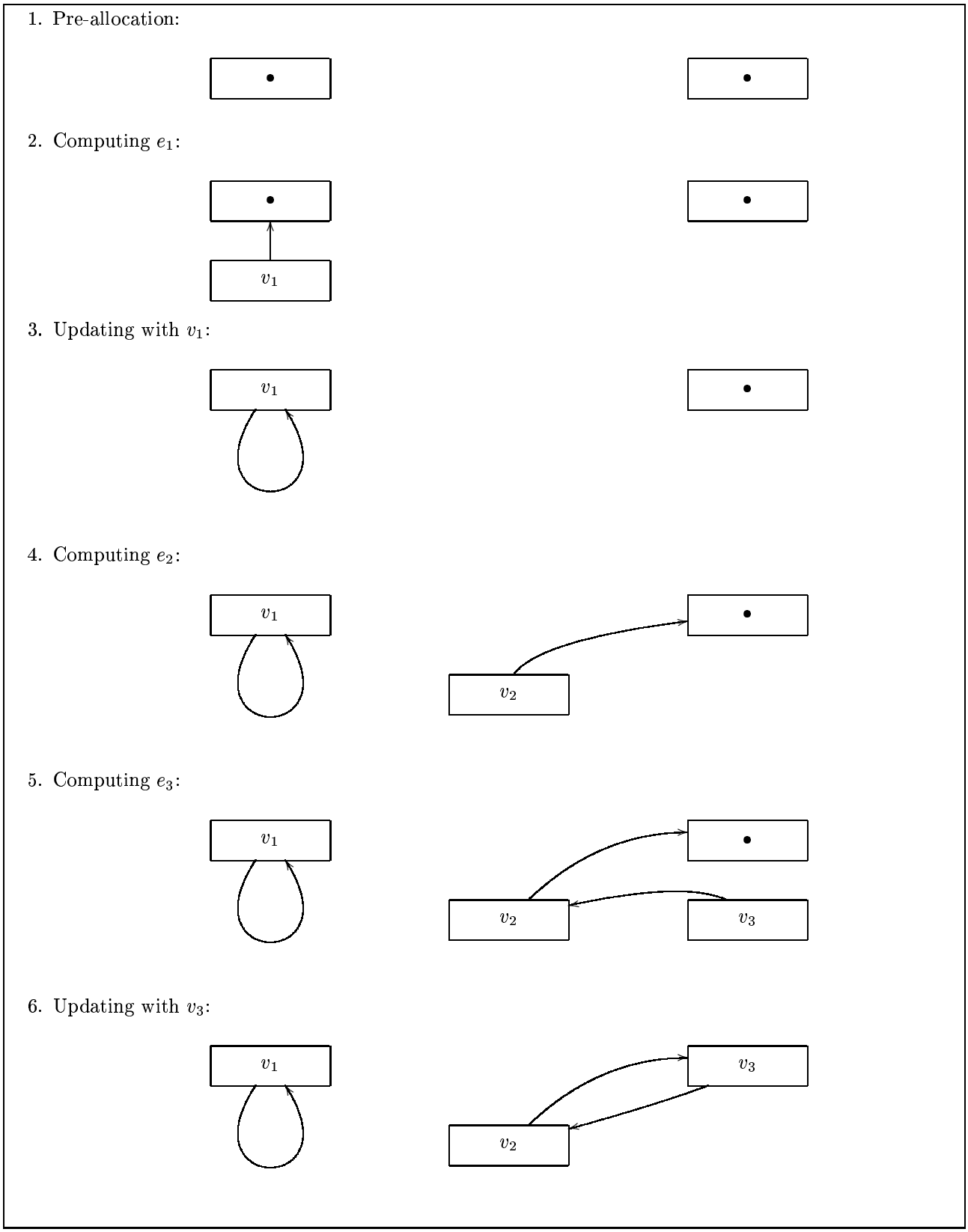


Figure 2: The refined "in-place updating trick"

$$
\begin{array}{rll}
x & \in \mathbf{Vars} & \text{Variable} \\
X & \in \mathbf{Names} & \text{Name} \\
\text{Expression:} \quad e \in \mathbf{expr} & ::= x \mid \lambda x.e \mid e_1 e_2 & \\
& \mid \{X_1 = e_1 \dots X_n = e_n\} \mid e.X & \\
& \mid \mathbf{let\ rec}\ x_1 = e_1 \dots x_n = e_n\ \mathbf{in}\ e &
\end{array}
$$

Figure 3: Syntax of $\lambda_\circ$

- More meta-variables:

$s ::= X_1 = e_1 \dots X_n = e_n$ Record
$b ::= x_1 = e_1 \dots x_n = e_n$ Binding

- Notations:

For a finite map f, and a set of variables P,
$\mathbf{dom}(f)$ is its domain, $\mathbf{cod}(f)$ is its codomain
$f_{|P}$ is its restriction to P, and $f_{\setminus P}$ is its restriction to $\mathbf{Vars} \setminus P$.

- Expressions of predictable shape:
$e_\downarrow \in \mathbf{Predictable} ::= \{o\} \mid \langle \iota \mid o \rangle \mid \mathbf{let\ rec}\ b\ \mathbf{in}\ e_\downarrow$

Figure 4: Meta-variables and notations

3 The source language $\lambda_\circ$

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. Variables are used as binders, as usual. Names are used for accessing record fields, as an external interface to other parts of the expression. Figure 4 recapitulates the meta-variables and notations we introduce in the remainder of this section. The syntax includes the λ-calculus constructs; variables x, abstraction $\lambda x.e$, and application $e_1 e_2$. The language also includes records $\{X_1 = e_1 \dots X_n = e_n\}$, record selection $e.X$ and a **let rec** construct. A mutually recursive definition has the shape $\mathbf{let\ rec}\ x_1 = e_1 \dots x_n = e_n\ \mathbf{in}\ e$, where arbitrary expressions are syntactically allowed as the right-hand side of a definition.

Syntactic correctness Records $s = (X_1 = e_1 \dots X_n = e_n)$ and bindings $b = (x_1 = e_1 \dots x_n = e_n)$ 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.

Consider the **let rec** binding $b = (x_1 = e_1 \dots x_n = e_n)$. We say that there is a *forward reference* from x_i to x_j if $i \leq j$, and x_j occurs free in e_i.

Forward references in bindings are allowed only when they point to a certain class of expressions, the expressions of *predictable* shape. As a first approximation, we say that the shape of an expression is predictable if it is a structure, a record, or a binding followed by an expression of predictable shape. Formally $e_\downarrow \in \mathbf{Predictable} ::= \{o\} \mid \langle \iota \mid o \rangle \mid \mathbf{let\ rec}\ b\ \mathbf{in}\ e_\downarrow$.

Sequences Records and bindings are often considered as finite maps in the sequel. We refer to them collectively as sequences, and use the usual notions on finite maps, such as the domain **dom**, the codomain **cod**, the restriction $\cdot_{|P}$ to a set P, or the co-restriction $\cdot_{\setminus P}$ outside of a set P.

3.2 Structural equivalence

We consider the expressions equivalent up to alpha-conversion of binding variables in structures and **let rec** expressions. For this, we define the *structural contraction relation*, in figure 8, relying on notions defined just below.

$$
\begin{array}{rcl}
\mathbf{UnsafeNewNames}(x, \lambda x.e) & = & \mathbf{Capt}_x(e) \cup \mathbf{FV}(e) \\
\mathbf{UnsafeNewNames}(x, \mathbf{let\ rec}\ b\ \mathbf{in}\ e) & = & \Big(\quad \mathbf{FV}(\mathbf{let\ rec}\ b\ \mathbf{in}\ e) \\
 & & \cup \quad \mathbf{Capt}_x(e) \\
 & & \cup \quad \bigcup_{(y \diamond f) \in b} (\{y\} \cup \mathbf{Capt}_x(f))\Big) \\
 & & \setminus \quad \{x\}
\end{array}
$$

Figure 5: Unsafe new names in $\lambda_\circ$

$$
\begin{array}{rcl}
\mathbf{Capt}_x(\mathbf{let\ rec}\ b\ \mathbf{in}\ e) & = & \begin{cases} \bigcup_{y \in \mathbf{dom}(b)} (\{y\} \cup \mathbf{Capt}_x(b(y))) \cup \mathbf{Capt}_x(e) \\ \qquad\qquad \text{if } x \in \mathbf{FV}(\mathbf{let\ rec}\ b\ \mathbf{in}\ e) \\ \emptyset \qquad\qquad \text{otherwise} \end{cases} \\
\mathbf{Capt}_x(x) = \mathbf{Capt}_x(\mathbf{c}) & = & \emptyset \\
\mathbf{Capt}_x(\lambda y.e) & = & \begin{cases} \{y\} \cup \mathbf{Capt}_x(e) \\ \quad \text{if } x \in \mathbf{FV}(\lambda y.e) \\ \emptyset \quad \text{otherwise} \end{cases}
\end{array}
$$

Other cases easy.

Figure 6: Capture in $\lambda_\circ$

Let $\sigma = \{x \mapsto y\}$.

$$
\begin{array}{rcl}
x\{\sigma\} & = & y \\
z\{\sigma\} & = & z \text{ if } z \neq x \\
\{X_1 = e_1 \ldots X_n = e_n\}\{\sigma\} & = & \{X_1 = e_1\{\sigma\} \ldots X_n = e_n\{\sigma\}\} \\
(\lambda z.e)\{\sigma\} & = & \begin{cases} \lambda z.(e\{\sigma\}) & \text{if } z \notin \{x, y\} \\ \lambda z.e & \text{otherwise} \end{cases} \\
(\mathbf{let\ rec}\ b\ \mathbf{in}\ e)\{\sigma\} & = & \begin{cases} \mathbf{let\ rec}\ b\{\sigma\}\ \mathbf{in}\ e\{\sigma\} & \text{if } \{x, y\} \cap \mathbf{dom}(b) = \emptyset \\ \mathbf{let\ rec}\ b\ \mathbf{in}\ e & \text{otherwise} \end{cases} \\
(x_1 = e_1 \ldots x_n = e_n)\{\sigma\} & = & (x_1 = e_1\{\sigma\} \ldots x_n = e_n\{\sigma\})
\end{array}
$$

Other cases easy.

Figure 7: Variable renaming in $\lambda_\circ$

A binder x, in a **let rec** or in a function, may be renamed into a new variable y, provided y meets some freshness conditions. Variable renaming is formally defined in figure 7, using notions defined in figures 5 and 6. Variable renaming is a total function, from pairs of an expression and a variable renaming $x \mapsto y$ (x is replaced with y), to expressions. In case renaming crosses a node binding one of the two variables x and y, it stops. Otherwise, it is propagated as usual. Therefore, variable renaming sometimes does not preserve meaning. For instance, renaming x with y in $\lambda y.x$ yields the same expression, since renaming does not cross the node binding y. This is why we introduce the notion of unsafe new names. It is defined in figure 5. A new name can be unsafe for a binder if it is captured by binders inside the sub-expression, as y is in the above example. The notion of capture is formalized by the **Capt** function in figure 6. Basically, $\mathbf{Capt}_x(e)$ denotes the set of binding variables located above occurrences of x in e. For instance $\mathbf{Capt}_x(\lambda y.x)$ is the set $\{y\}$. A new name can also be unsafe for a binder when it is free in the considered sub-expression. For example, renaming x to y in $\lambda x.(xy)$ does not preserve meaning. The structural contraction relation, $\rightsquigarrow_s$, defined in figure 8, allows to rename a binder, provided the corresponding variable renaming is correct on the considered expression. The structural reduction relation $\dashrightarrow_s$ is the contextual closure of the structural contraction relation. These two relations are symmetric, and therefore the transitive closure $\dashrightarrow_s^*$ of $\dashrightarrow_s$ is a congruence, called the structural equivalence relation, and also written $=_s$.

In the following, all expressions are considered up to structural equivalence $=_s$.

$$\frac{y \notin \mathbf{UnsafeNewNames}(x, \mathbf{let\ rec}\ b_1, x = e, b_2\ \mathbf{in}\ f) \qquad \sigma = x \mapsto y}{\mathbf{let\ rec}\ b_1, x = e, b_2\ \mathbf{in}\ f \rightsquigarrow_s \mathbf{let\ rec}\ b_1\{\sigma\}, y = e\{\sigma\}, b_2\{\sigma\}\ \mathbf{in}\ f\{\sigma\}}$$

$$\frac{y \notin \mathbf{UnsafeNewNames}(x, \lambda x.e)}{\lambda x.e \rightsquigarrow_s \lambda y.(e\{x \mapsto y\})}$$

Figure 8: Structural contraction relation of $\lambda_\circ$

Configuration: $c ::= b \vdash e$

Value: $v \in \mathbf{values} ::= x \mid \lambda x.e \mid \{s_v\}$

Answer: $a \in \mathbf{answers} ::= b_v \vdash v$

More meta-variables:

$s_v ::= X_1 = v_1 \ldots X_n = v_n$ Value record

$b_v ::= x_1 = v_1 \ldots x_n = v_n$ Value binding

Figure 9: Configurations and results in $\lambda_\circ$

3.3 Semantics

The semantics of $\lambda_\circ$ is quite standard, except for what concerns **let rec** bindings.

As shown in figure 9, values include functions $\lambda x.e$ and records of values $\{s_v\}$, where s_v denotes an evaluated record $X_1 = v_1 \ldots X_n = v_n$.

The semantics of record selection and of function application are defined in figure 10, by *computational contraction* rules, defining the local *computational contraction relation* $\rightsquigarrow_c$. Record projection selects the appropriate field in the record ; and the application of a function $\lambda x.e$ to a value v reduces to the body of the function, where the argument has been bound to x.

Five operations are necessary for handling bindings properly, all defined Ariola et al. [2].

1. A first operation is **let rec** lifting. It consists in lifting a **let rec** node up one level in an expression. For example, an expression of the shape $e_1 + (\mathbf{let\ rec}\ b\ \mathbf{in}\ e_2)$ becomes $\mathbf{let\ rec}\ b\ \mathbf{in}\ e_1 + e_2$.

2. A second operation is 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 into the current one. An expression of the shape $\mathbf{let\ rec}\ b_1, x = (\mathbf{let\ rec}\ b_2\ \mathbf{in}\ e), b_3\ \mathbf{in}\ f$ becomes $\mathbf{let\ rec}\ b_1, b_2, x = e, b_3\ \mathbf{in}\ f$, provided no variable capture occurs.

3. A third operation is external merging. The shape of results 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 $\mathbf{let\ rec}\ b_1\ \mathbf{in}\ \mathbf{let\ rec}\ b_2\ \mathbf{in}\ e$ becomes $\mathbf{let\ rec}\ b_1, b_2\ \mathbf{in}\ e$, provided no variable capture occurs.

4. A fourth operation, external substitution, allows to access bound variables when defined by a surrounding binding. An expression of the shape $\mathbf{let\ rec}\ b\ \mathbf{in}\ \mathbb{C}[x]$ becomes $\mathbf{let\ rec}\, b\, \mathbf{in}\, \mathbb{C}[e]$, if $x = e$ appears in b and x is not captured by $\mathbb{C}$, and no variable capture occurs.

5. A last operation, 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 $\mathbf{let\ rec}\ b_1, y = \mathbb{C}[x], b_2\ \mathbf{in}\ e$ becomes $\mathbf{let\ rec}\, b_1, y = \mathbb{C}[f], b_2\, \mathbf{in}\, e$ if x is defined as f in b_1, and not captured by $\mathbb{C}$, and no variable capture occurs.

The question is how to arrange these operations to make the evaluation deterministic and to ensure that it reaches the result when it exists. Our choice can be summed up as follows. There is a topmost binding. When this topmost binding is already evaluated, evaluation can proceed under this binding. Otherwise, evaluation is

• **Contraction rules**

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

$$\frac{\mathbf{dom}(b) \perp \mathbf{FV}(\mathbb{L})}{\mathbb{L}[\textbf{let rec } b \textbf{ in } e] \rightsquigarrow_c \textbf{let rec } b \textbf{ in } \mathbb{L}[e]} \quad \text{(Lift)}$$

• **Computational reduction rules**

$$\frac{e \rightsquigarrow_c e'}{\mathbb{E}[e] \dashrightarrow_c \mathbb{E}[e']} \quad \text{(Context)} \qquad \frac{\mathbb{E}[\mathbb{N}](x) = v}{\mathbb{E}[\mathbb{N}[x]] \dashrightarrow_c \mathbb{E}[\mathbb{N}[v]]} \quad \text{(Subst)}$$

$$\frac{\mathbf{dom}(b) \perp (\mathbf{dom}(b_v) \cup \mathbf{FV}(b_v))}{(b_v \vdash \textbf{let rec } b \textbf{ in } e) \dashrightarrow_c b_v, b \vdash e} \quad \text{(EM)}$$

$$\frac{\mathbf{dom}(b_1) \perp \{x\} \cup \mathbf{dom}(b_v, b_2) \cup \mathbf{FV}(b_v, b_2) \cup \mathbf{FV}(f)}{(b_v, x = (\textbf{let rec } b_1 \textbf{ in } e), b_2 \vdash f) \dashrightarrow_c (b_v, b_1, x = e, b_2 \vdash f)} \quad \text{(IM)}$$

• **Evaluation contexts**

Lift context:
$\mathbb{L} ::= \Box e \mid v\Box \mid \Box.X \mid \{\mathbb{S}\}$
Multiple lift context:
$\mathbb{F} ::= \Box \mid \mathbb{L}[\mathbb{F}]$
Evaluation context:
$\mathbb{E} ::= (b_v \vdash \mathbb{F}) \mid (\mathbb{B}[\mathbb{F}] \vdash e)$

Record contexts:
$\mathbb{S} ::= s_v, X = \Box, s$
Sequence contexts:
$\mathbb{B} ::= b_v, x = \Box, b$
Strict contexts:
$\mathbb{N} ::= \Box v \mid \Box.X$

• **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 10: Reduction semantics for $\lambda_\circ$

allowed inside this binding. If evaluation meets another binding inside the expression, this binding is lifted to be immediately under the topmost binding. Then, it is merged with the latter, internally or externally according to the context. External and internal substitutions are allowed only from the evaluated part of the topmost binding. In order to simplify the presentation of the translation and the correctness proof, we distinguish this topmost binding syntactically : the global *computational reduction relation* $\dashrightarrow_c$ is a binary relation on *configurations* c, which are pairs of a binding, the topmost binding, and an expression, written $b \vdash e$ (see figure 9). Here, the topmost binding is close to the usual notion of runtime environment, with the additional feature that bound values can be mutually recursive.

More formally, **let rec** handling is done through one additional computational contraction rule Lift performing the lifting operation, and a *computational reduction relation*, defined in figure 10.

The contraction rule Lift lifts a **let rec** binding up a *lift context*. As defined in figure 10, a lift context is any non-**let rec** expression, where the special context hole variable $\Box$ appears immediately under the first node, in position of the next sub-expression evaluated.

The second contraction rule IM corresponds to internal merging. If, during the evaluation of the topmost binding, one definition evaluates to a binding, then this binding is merged with the topmost one. The evaluation can then continue.

The computational reduction relation extends the computational contraction relation to any evaluation context, as defined in figure 10. We call a multiple lift context a series of nested lift contexts, and an evaluation context is a multiple lift context, possibly inside a partially evaluated binding, or under a fully evaluated binding.

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

$x \in \mathbf{Vars}$
$X \in \mathbf{Names}$

Expression:

$E \in \mathbf{Expr} ::= x \mid \lambda x.E \mid EE$		λ-calculus
$\mid \mathbf{let}\ x_1 = E_1 \ldots x_n = E_n\ \mathbf{in}\ E$		Non-recursive **let** binding
$\mid \{X_1 = E_1 \ldots X_n = E_n\} \mid E.X$		Records
$\mid l \mid \mathbf{alloc} \mid \mathbf{update}$		Locations, allocation, mutation

Figure 11: Syntax of λ_{alloc}

Configuration:

$C ::= \Theta \vdash E$
$\Theta \in \mathbf{Heaps} = \mathbf{Vars} \xrightarrow{\text{Fin}} \mathbf{HeapValues}$

Answer:

$A \in \mathbf{Answers} ::= \Theta \vdash V$
$V \in \mathbf{Values} ::= x \mid l$

More meta-variables:

$H_v \in \mathbf{HeapValues} ::= \lambda x.E \mid \mathbf{alloc}\ n \mid \{S_v\}$
$S_v ::= X_1 = V_1 \ldots X_n = V_n$
$B ::= x_1 = E_1 \ldots x_n = E_n$

Figure 12: Configurations and results in λ_{alloc}

Finally, the external and internal substitution operations are modeled within a single reduction rule SUBST. This rule transforms an expression of the shape $\mathbb{E}[\mathbb{N}[x]]$ into $\mathbb{E}[\mathbb{N}[v]]$, provided the context $\mathbb{E}[\mathbb{N}]$ defines x as v and no variable capture occurs. The meta-variable $\mathbb{N}$ ranges over *strict* contexts. A strict context is a context that requires a non-variable node to evaluate. An example of strict context is $\Box v$, that is, the function part of a function application. An example of a non-strict context is $(\lambda x.e)\Box$, that is, the argument part of a function application, where a variable would allow the evaluation to continue. Strict contexts are formally defined in figure 10. The SUBST rule replaces a variable in a strict context with its value, according to the context. As indicated in figure 10, evaluation contexts define the variable they bind, in two possible ways. First, a topmost, semantically correct, fully evaluated **let rec** binding defines the variables it binds for the nodes under it. Second, if $(b_v, x \diamond \mathbb{F}, b)$ is the topmost, partially evaluated binding, then b_v defines the variables it binds, inside $\mathbb{F}$, and later inside b. The two rules defining access in evaluation contexts in figure 10 show how these definitions may be used. The two different ways of access correspond to the external and internal substitution operations, respectively.

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

4 The target language λ_{alloc}

The syntax of the target language λ_{alloc} is presented in figure 11. It distinguishes variables x from names X. It includes the constructs of the λ-calculus (function abstraction and application) and a non recursive **let** binding. Additionally, there are constructs for record operations (construction and selection), and constructs for modeling the heap: an allocation operator **alloc**, an update operator **update**, and locations l.

The semantics of λ_{alloc} is defined as a structural reduction relation on *configurations*. As defined in figure 12, a configuration is a pair of a heap and an expression. A heap is a finite map from locations l to evaluated

• **Substitutions** Let $\sigma \in \mathbf{Subst} = \mathbf{Vars} \xrightarrow{\text{Fin}} \mathbf{Values}$.

$$\begin{array}{rcl} z\{\sigma\} & = & \sigma(z) \\ (\lambda x.E)\{\sigma\} & = & \lambda x.(E\{\sigma_{\setminus\{x\}\cup\sigma^{-1}(\{x\})}\}) \\ (\textbf{let } B \textbf{ in } F)\{\sigma\} & = & \textbf{let } B\{\sigma\} \textbf{ in } F\{\sigma_{\setminus \mathbf{dom}(B)\cup\sigma^{-1}(\mathbf{dom}(B))}\} \\ (x = E, B)\{\sigma\} & = & (x = E\{\sigma\}, B\{\sigma_{\setminus\{x\}\cup\sigma^{-1}(\{x\})}\}) \\ \Theta \vdash E\{\sigma\} & = & \sigma \circ \Theta \vdash E\{\sigma\} \\ & & \text{Other cases easy.} \end{array}$$

• **Capture**

$$\begin{array}{rcl} \mathbf{Capt}_x(\textbf{let } y = E, B \textbf{ in } F) & = & \mathbf{Capt}_x(\lambda y.\textbf{let } B \textbf{ in } F) \cup \mathbf{Capt}_x(E) \\ \mathbf{Capt}_x(x) & = & \emptyset \\ \mathbf{Capt}_x(\lambda y.E) & = & \begin{cases} \{y\} \cup \mathbf{Capt}_x(E) \\ \quad \text{if } x \in \mathbf{FV}(\lambda y.E) \\ \emptyset \qquad \text{otherwise} \end{cases} \end{array}$$

Other cases easy.

• **Unsafe new names**

$$\begin{array}{rcl} \mathbf{UnsafeNewNames}(x, \lambda x.E) & = & \mathbf{Capt}_x(E) \cup \mathbf{FV}(E) \\ \mathbf{UnsafeNewNames}(x, \textbf{let } x = E, B \textbf{ in } F) & = & \mathbf{UnsafeNewNames}(x, \lambda x.(\textbf{let } B \textbf{ in } F)) \end{array}$$

• **Structural reduction**

$$\frac{y \notin \mathbf{UnsafeNewNames}(x, \lambda x.E)}{\lambda x.E \rightsquigarrow_s \lambda y.(E\{x \mapsto y\})} \qquad \frac{y \notin \mathbf{UnsafeNewNames}(x, \textbf{let } x = E, B \textbf{ in } F)}{\textbf{let } x = E, B \textbf{ in } F \rightsquigarrow_s \textbf{let } y = E, (B\{x \mapsto y\}) \textbf{ in } F\{x \mapsto y\}}$$

Figure 13: Structural equivalence in λ_{alloc}

heap blocks. An evaluated heap block $H_v \in \mathbf{HeapValues}$ is either a function $\lambda x.E$, or an evaluated record $\{S_v\}$ (where $Sv ::= X_1 = V_1 \ldots X_n = V_n$), or an application of the shape $\mathbf{alloc}\, n$, for $n \in \mathbb{N}$. Such applications model dummy heap blocks, containing unspecified data. A *well-formed* configuration is such that all the locations mentioned are bound in its heap.

Evaluated heap blocks are not values. Only 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 result of the evaluation of $\lambda x.E$ is a configuration $\Theta \vdash l$, where the location l is bound to $\lambda x.[\![e]\!]$ in the heap Θ.

The related operators in the language are **alloc**, which creates a new empty block of size given by its argument, and **update**, which copies its second argument in place of its first one, provided they have the same size. For this, we assume given a function **Size** from λ_{alloc} heap value blocks to $\mathbb{N}$.

Notation We write $\Theta\langle l \mapsto H_v\rangle$ 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 $\Theta + \{l \mapsto H_v\}$ to denote the union of Θ and $\{l \mapsto H_v\}$.

4.1 Structural equivalence

In λ_{alloc}, a notion of structural equivalence identifies expressions modulo variable and location renaming. Locations are bound only by heaps, at toplevel in configurations. We consider configurations equal modulo renaming of bound locations. This relation is easy to define since the location renaming never cross any location binder, so we do not formalize it here. However, we have to define the structural equivalence modulo variable renaming. A binder x, in a **let** or in a function, may be renamed into a new variable y, provided y meets some freshness conditions. Structural equivalence is formally defined in figure 13.

Substitutions First, variable renaming is defined. It is a total function, from pairs of an expression and a variable renaming $x \mapsto y$ (x is replaced with y), to expressions. Nevertheless, we will see that the computational

$$\frac{\Theta(l) = \lambda x.E}{\Theta \vdash lV \rightsquigarrow_c \Theta \vdash E\{x \mapsto V\}} \quad (\textsc{Beta}) \qquad \frac{l \notin \mathbf{dom}(\Theta)}{\Theta \vdash H_v \rightsquigarrow_c \Theta + \{l \mapsto H_v\} \vdash l} \quad (\textsc{Allocate})$$

$$\frac{\Theta(l) = \{S_v\}}{\Theta \vdash l.X \rightsquigarrow_c \Theta \vdash S_v(X)} \quad (\textsc{Project}) \qquad \frac{\mathbf{Size}(\Theta(l_1)) = \mathbf{Size}(\Theta(l_2))}{\Theta \vdash \mathbf{update}\, l_1\, l_2 \rightsquigarrow_c \Theta\langle l_1 \mapsto \Theta(l_2)\rangle \vdash \{\}} \quad (\textsc{Update})$$

$$\frac{\mathbf{dom}(B) \perp \Lambda}{\Theta \vdash \Lambda[\mathbf{let}\ \ B\ \ \mathbf{in}\ \ E] \rightsquigarrow_c \Theta \vdash \mathbf{let}\ \ B\ \ \mathbf{in}\ \ \Lambda[E]} \quad (\textsc{Lift})$$

Figure 14: Computational contraction rules for λ_{alloc}

reduction relation uses a more complex notion of substitution than just variable renaming: it must also replace variables with locations in some cases. Therefore, substitutions are elements of $\mathbf{Subst} = \mathbf{Vars} \xrightarrow{\text{Fin}} \mathbf{Values}$. We interpret them as total functions from variables to values, extending them with the identity function on variables, outside of their syntactic domain. The domain $\mathbf{dom}(\sigma)$ of a substitution σ is the set of variables x such that $\sigma(x) \neq x$. We sometimes consider substitutions as sets, taking the union of two of them when it makes sense, and sometimes we compose them, in the reverse notation, since they come from the right. The composition of σ_1 and σ_2 is defined by $x\{\sigma_1 \circ \sigma_2\} = x\{\sigma_1\}\{\sigma_2\}$: it acts as σ_1, then 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, as described in figure 13 (where we take the usual notation for substitution $E\{\sigma\}$, meaning $\sigma(E)$). In case the substitution crosses a binder x, then it forgets any information about x. Thus, under this binder the substitution becomes $\sigma_{\setminus\{x\} \cup \sigma^{-1}(\{x\})}$. Otherwise, it is propagated as usual. Therefore, substitution sometimes does not preserve meaning. For instance, renaming x with y in $\lambda y.x$ yields the same expression, since substitution does not cross the node binding y.

Structural equivalence This is why we introduce the notion of unsafe new names. It is defined in figure 5. A new name can be unsafe for a binder if it is captured by binders inside the sub-expression, as y is in the above example. The notion of capture is formalized by the **Capt** function in figure 13. Basically, $\mathbf{Capt}_x(e)$ denotes the set of binding variables located above occurrences of x in e. For instance $\mathbf{Capt}_x(\lambda y.x)$ is the set $\{y\}$. A new name can also be unsafe for a binder when it is free in the considered sub-expression. As an example, renaming x to y in $\lambda x.(xy)$ does not preserve meaning.

The structural contraction relation, $\rightsquigarrow_s$, defined in figure 13, allows to rename a binder, provided the corresponding variable renaming is correct on the considered expression. The structural reduction relation $\dashrightarrow_s$ is the contextual closure of the structural contraction relation. These two relations are symmetric, and therefore the transitive closure $\dashrightarrow_s^*$ of $\dashrightarrow_s$ is a congruence, called the structural equivalence relation, and also written $=_s$.

4.2 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 constructors and a computational reduction relation that handles global rules. As in $\lambda_\circ$, evaluation results are values surrounded by a heap binding:

$$A \in \mathbf{Answers} ::= \Theta \vdash V.$$

Computational contraction relation The computational contraction relation is defined by the rules in figure 14, using the notion of lift contexts in figure 15.

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

Lift context:

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

Record context:

$$\Sigma ::= S_v, X = \Box, S$$

Multiple lift context:

$$\Phi ::= \Box \mid \Lambda[\Phi]$$

Figure 15: Evaluation contexts of λ_{alloc}

$$\frac{\Theta \vdash E \rightsquigarrow_c \Theta' \vdash E'}{\Theta \vdash \Phi[E] \dashrightarrow_c \Theta' \vdash \Phi[E']} \text{ (CONTEXT)} \qquad \Theta \vdash \textbf{let } x = V, B \textbf{ in } E \dashrightarrow_c \Theta \vdash (\textbf{let } B \textbf{ in } E)\{x \mapsto V\} \text{ (LET)}$$

$$\Theta \vdash \textbf{let } \epsilon \textbf{ in } E \dashrightarrow_c \Theta \vdash E \text{ (EMPTYLET)} \qquad \frac{l \notin (\mathbf{FV}(\Theta_{\setminus\{l\}}) \cup \mathbf{dom}(\Theta_{\setminus\{l\}}) \cup \mathbf{FV}(E))}{\Theta \vdash E \dashrightarrow_c \Theta_{\setminus\{l\}} \vdash E} \text{ (GC)}$$

$$\Theta \vdash \textbf{let } B_1 \textbf{ in let } B_2 \textbf{ in } E \dashrightarrow_c \Theta \vdash \textbf{let } B_1, B_2 \textbf{ in } E \text{ (EM)}$$

Figure 16: Computational reduction in λ_{alloc}

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

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: $\Theta + \{l \mapsto H_v\} \vdash l$ (l fresh). If H_v is a dummy block $\textbf{alloc}\, n$, the result is a dummy block on the heap.

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

Finally, as in $\lambda_\circ$, the evaluation of bindings is confined to the toplevel of terms, whence the LIFT rule, which lifts a binding outside of a lift context. In λ_{alloc}, lift contexts are of the shape

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

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

Computational reduction relation The computational reduction relation is defined in figure 16.

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

The LET rule describes the toplevel evaluation of bindings. Once the first definition is evaluated, the binding variable is replaced with the obtained value in the rest of the expression. Eventually, when the binding is empty, it 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.

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

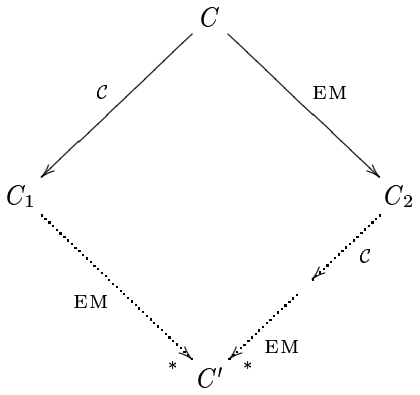
4.3 The λ_{alloc} calculus and its confluence

The set of *terms* of the λ_{alloc} calculus is the set of equivalence classes for $=_s$. The computational reduction relation on expressions is compatible with $=_s$, so we may extend it to terms, to obtain the reduction relation $\longrightarrow$.

Definition 1 *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 outcomes when two successive bindings are encountered. It is therefore important to make sure that λ_{alloc} is confluent. Let $\xrightarrow{\mathcal{C}}$ be the relation defined by the rules Context, Let, and EmptyLet. It is syntax directed, and therefore deterministic.

We first prove the following proposition, which is also described by the following diagram, where the plain arrows are universally quantified, and the dotted ones are existentially quantified.



Proposition 1 *For all configurations C, C_1, and C_2 such that $C \xrightarrow{\mathcal{C}} C_1$ and $C \xrightarrow{\mathrm{EM}} C_2$, there exists a configuration C' such that $C_1 \xrightarrow{\mathrm{EM}^*} C'$ and $C_2 \xrightarrow{\mathcal{C}}\xrightarrow{\mathrm{EM}^*} C'$.*

Proof

If $C \xrightarrow{\textsc{EmptyLet}} C_1$, the two obtained configurations are identical. If $C \xrightarrow{\mathcal{C}} C_1$ by rule Let, then the two reductions simply commute. If $C \xrightarrow{\textsc{Context}} C_1$, then we have to examine the underlying contraction step $C \rightsquigarrow_c C_1$. In all cases but one, the two reduction steps simply commute. The only problematic case is when the applied rule is Lift. We have $C = \Theta \vdash \Phi[E]$, with $E = \Lambda[\mathbf{let}\ B\ \mathbf{in}\ E_1]$, and $C_1 = \Theta \vdash \Phi[\mathbf{let}\ B\ \mathbf{in}\ \Lambda[E_1]]$.

- If $\Lambda ::= \Box F \mid V\Box \mid \{\Sigma\} \mid \Box.X$, as rule EM applies on C, we must have Φ of the shape $\mathbf{let}\ x = \Phi_1, B_1\ \mathbf{in}\ \mathbf{let}\ B_2\ \mathbf{in}\ F'$. Therefore

  $$C_1 = \Theta \vdash \mathbf{let}\ x = \Phi_1[\mathbf{let}\ B\ \mathbf{in}\ \Lambda[E_1]], B_1\ \mathbf{in}\ \mathbf{let}\ B_2\ \mathbf{in}\ F',$$

  and

  $$C_2 = \Theta \vdash \mathbf{let}\ x = \Phi_1[\Lambda[\mathbf{let}\ B\ \mathbf{in}\ E_1]], B_1, B_2\ \mathbf{in}\ F'.$$

  Let

  $$C' = \Theta \vdash \mathbf{let}\ x = \Phi_1[\mathbf{let}\ B\ \mathbf{in}\ \Lambda[E_1]], B_1, B_2\ \mathbf{in}\ F'.$$

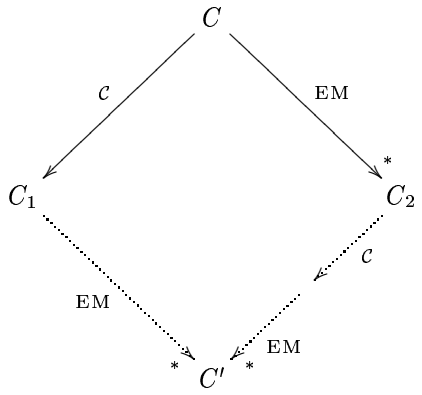
  We obtain easily that C_1 and C_2 both reduce to C', in one step of $\xrightarrow{\mathrm{EM}}$ and $\xrightarrow{\textsc{Lift}}$, respectively, which is as expected.

- If $\Lambda = \mathbf{let}\ x = \Box, B_1\ \mathbf{in}\ F$, then Φ might still be of the shape $\mathbf{let}\, x = \Phi_1, B_1' \,\mathbf{in}\, \mathbf{let}\, B_2 \,\mathbf{in}\, F'$, in which case the previous reasoning applies. If it is not of this shape, then the **let** binding contained in Λ is part of the EM redex, so $\Phi = \Box$, and F is of the shape $\mathbf{let}\ B_2\ \mathbf{in}\ F''$. So, we have a diagram of the shape:

$$\begin{array}{lcl}
\Theta \vdash \mathbf{let}\, x = (\mathbf{let}\ B\ \mathbf{in}\ E_1), B_1 \,\mathbf{in}\, \mathbf{let}\, B_2 \,\mathbf{in}\, F'' & \xrightarrow{\textsc{Lift}} & \Theta \vdash \mathbf{let}\, B \,\mathbf{in}\, \mathbf{let}\, x = E_1, B_1 \,\mathbf{in}\, \mathbf{let}\ B_2\ \mathbf{in}\ F'' \\
\Big\downarrow \mathrm{EM} & & \Big\downarrow \mathrm{EM} \\
\Theta \vdash \mathbf{let}\ x = (\mathbf{let}\ B\ \mathbf{in}\ E_1), B_1, B_2\ \mathbf{in}\ F'' & & \Theta \vdash \mathbf{let}\, B, x = E_1, B_1 \,\mathbf{in}\, \mathbf{let}\ B_2\ \mathbf{in}\ F'' \\
\Big\downarrow \textsc{Lift} & & \Big\downarrow \mathrm{EM} \\
\Theta \vdash \mathbf{let}\, B \,\mathbf{in}\, \mathbf{let}\, x = E_1, B_1, B_2 \,\mathbf{in}\, F'' & \xrightarrow{\mathrm{EM}} & \Theta \vdash \mathbf{let}\, B, x = E_1, B_1, B_2 \,\mathbf{in}\, F''
\end{array}$$

□

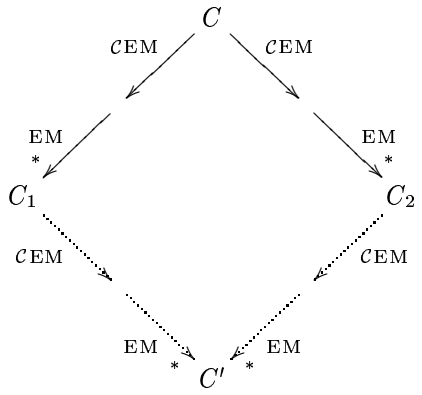
This result extends by a simple induction to the following corollary, pictorially described by the following diagram.



Corollary 1 *For all configurations C, C_1, and C_2 such that $C \xrightarrow{\mathcal{C}} C_1$ and $C \xrightarrow{\mathrm{EM}^*} C_2$, there exists a configuration C' such that $C_1 \xrightarrow{\mathrm{EM}^*} C'$ and $C_2 \xrightarrow{\mathcal{C}}\xrightarrow{\mathrm{EM}^*} C'$.*

Then, the relation $\xrightarrow{\mathcal{C}\mathrm{EM}}$ is defined as $\xrightarrow{\mathcal{C}}$, extended with rule EM. Formally, $\xrightarrow{\mathcal{C}\mathrm{EM}} = \xrightarrow{\mathcal{C}} \cup \xrightarrow{\mathrm{EM}}$.

Thanks to the previous corollary, we prove that the $\xrightarrow{\mathcal{C}\mathrm{EM}}$ relation is confluent. This is done by considering the relation $\xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*}$, which is strongly confluent. In other terms for any two reduction steps $C \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*} C_1$ and $C \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*} C_2$, there exist a configuration C' and two reduction steps $C_1 \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*} C'$ and $C_2 \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*} C'$. A pictorial view of this is given by the following diagram:

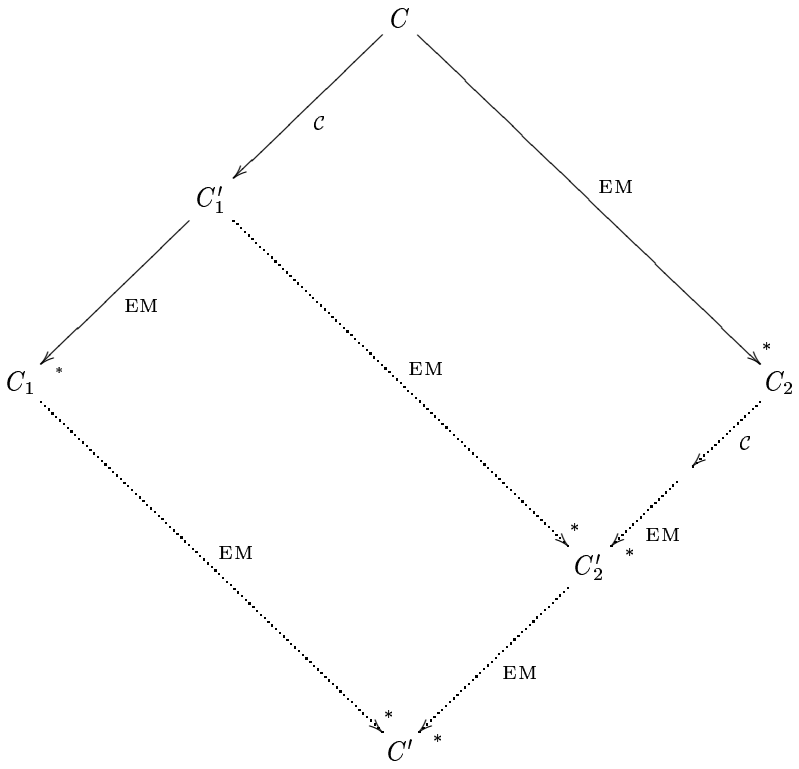


Proposition 2 *The relation $\xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*}$ is strongly confluent.*

Proof To prove this last statement, we proceed by case on the $\mathcal{C}$EM rules applied, from C, to reach C_1 and C_2, respectively. If the two rules are EM, then as this relation is deterministic, we conclude easily, and similarly if the two reductions are $\xrightarrow{\mathcal{C}}$ steps. The only relevant case is when one reduction is a $\xrightarrow{\mathcal{C}}$ step, say $C \xrightarrow{\mathcal{C}} C_1$, and the other is in $\xrightarrow{\mathrm{EM}}$.

In this case, we have $C \xrightarrow{\mathcal{C}} C_1' \xrightarrow{\mathrm{EM}^*} C_1$. By the previous corollary, we obtain a C_2' such that $C_2 \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*} C_2'$. Then, by confluence of the deterministic relation $\xrightarrow{\mathrm{EM}}$, we obtain C' such that $C_1' \xrightarrow{\mathrm{EM}^*} C'$ and $C_2' \xrightarrow{\mathrm{EM}^*} C'$. This configuration is also such that C_1 and C_2 reduce to it by relation $\xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}^*}$, in at most one step.

This is depicted by the following diagram.



□

Corollary 2 (Confluence of λ_{alloc}) *The λ_{alloc} calclulus is confluent.*

5 Translation

5.1 Generalized contexts in λ_{alloc}

The purpose of this paper is to prove that $\lambda_\circ$ can be faithfully translated into λ_{alloc}. A desired property for this translation, in order to make the proof of correctness easier, is that a result is translated as a result, not needing any additional computation. However, a simple abstraction such as $\lambda x.x$ is a value of $\lambda_\circ$, and could be translated as such in λ_{alloc}, but is not a result of λ_{alloc}. The correct translation is rather the configuration $\{l \mapsto \lambda x.x\} \vdash l$. The drawback of such a method is that the translation is no longer compositional, at least in the usual sense. Indeed, 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.

5.1.1 Definition

In order to overcome this difficulty, we introduce a non-standard notion of contexts in λ_{alloc}, which take as an argument configurations, rather than just expressions. Configurations are pairs of a heap and a multiple lift context, and the application of a context $\Theta \vdash \Phi$ to a configuration $\Theta' \vdash E$ is $\Theta + \Theta' \vdash \Phi[E]$.

We are not done yet. We have indeed seen that results in $\lambda_\circ$ can be of the shape $b_v \vdash v$. We imagine that b_v will be translated as the heap, roughly. But heaps of λ_{alloc} only contain heap blocks, i.e. dummy blocks, functions or evaluated records. Therefore, in the case where b_v contains definitions of the shape $x = y$ for example (or $x = 1$ if we had constants), we have to find another solution. Furthermore, this solution has to take into account the asymmetry of **let rec** in $\lambda_\circ$. Indeed, the heap $x = y, z = x$ in fact 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 $\{z \mapsto x\} \circ \{x \mapsto y\}$ (recall that composition of substitution is "left to right").

But then, contexts again become a bit more complicated, because they must include a substitution part. Indeed, the $\lambda_\circ$ context $x = y, z = x \vdash \Box$ 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 them $\Theta \vdash \Phi[\sigma]$, 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, in the following sense.

Definition 2 (Substitution correctness) *A substitution σ is correct for an expression E iff*

$$\forall x \in \mathbf{dom}(\sigma), \sigma(x) \notin \mathbf{Capt}_x(E).$$

This definition extends straightforwardly to heaps and configurations. 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.

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

5.1.2 Properties

In this section, we prove some properties of stability of the reduction relation inside contexts. Not every reduction step is valid inside contexts, since for instance the Let and EmptyLet are only valid at toplevel. However, we will see that inside contexts of the shape $\Theta \vdash \Box[\sigma]$, reduction is preserved.

We first prove that contraction is preserved under correct substitution.

Proposition 3 *If $C_1 \rightsquigarrow_c C_2$ and σ is correct for C_1 and C_2, then $C_1\{\sigma\} \rightsquigarrow_c C_2\{\sigma\}$.*

Proof By case on the applied contraction rule. Let $C_i = \Theta_i \vdash E_i$, for $i = 1, 2$.

Beta. Then $E_1 = lV$, and $\Theta_1 = \Theta_2$, and $\Theta_1(l) = \lambda x.E$. We have $E_1\{\sigma\} = l(V\{\sigma\})$, and as σ is correct, $(\lambda x.E)\{\sigma\} = \lambda x.(E\{\sigma\})$. So $\Theta_1\{\sigma\} \vdash l(V\{\sigma\}) \rightsquigarrow_c \Theta_2\{\sigma\} \vdash E\{\sigma\}\{x \mapsto (V\{\sigma\})\}$. As σ is correct for C_1, x is not in the domain or codomain of σ, so $\sigma \circ \{x \mapsto (V\{\sigma\})\} = \{x \mapsto V\} \circ \sigma$, and therefore $C_1\{\sigma\} \rightsquigarrow_c \Theta_2\{\sigma\} \vdash E_2\{\sigma\}$.

Allocate, Update, Project. Similar.

Lift We again have $\Theta_1 = \Theta_2$, with $E_1 = \Lambda[\mathbf{let}\ B\ \mathbf{in}\ E]$ and $E_2 = \mathbf{let}\ B\ \mathbf{in}\ \Lambda[E]$. By the side condition on the Lift rule, we also know that $\mathbf{dom}(B) \perp \mathbf{FV}(\Lambda)$. By hypothesis, we finally have $\mathbf{dom}(B)$ disjoint from the domain and codomain of σ.

So, $C_1\{\sigma\} = \Theta_1\{\sigma\} \vdash \Lambda\{\sigma\}[\mathbf{let}\ B\{\sigma\}\ \mathbf{in}\ E\{\sigma\}]$, which reduces to
$\Theta_1\{\sigma\} \vdash \mathbf{let}\, B\{\sigma\}\, \mathbf{in}\, \Lambda\{\sigma\}[E\{\sigma\}]$, as expected.

□

This property extends to computational reduction.

Proposition 4 *If $C_1 \longrightarrow C_2$ and σ is correct for C_1 and C_2, then $C_1\{\sigma\} \longrightarrow C_2\{\sigma\}$.*

Proof By case on the applied rule. Let again $C_i = \Theta_i \vdash E_i$, for $i = 1, 2$.

Context. By application of the previous proposition.

EmptyLet. Trivial.

Let. We have $C_1 = \Theta_1 \vdash \mathbf{let}\ x = V, B\ \mathbf{in}\ E$, and $C_2 = \Theta_1 \vdash \mathbf{let}\ B\{x \mapsto V\}\ \mathbf{in}\ E\{x \mapsto V\}$. So,

$$C_1\{\sigma\} = \Theta_1\{\sigma\} \vdash \mathbf{let}\ x = (V\{\sigma\}), B\{\sigma\}\ \mathbf{in}\ (E\{\sigma\}),$$

which reduces to

$$\Theta_1\{\sigma\} \vdash \mathbf{let}\ B\{\sigma\}\{x \mapsto (V\{\sigma\})\}\ \mathbf{in}\ (E\{\sigma\}\{x \mapsto (V\{\sigma\})\}),$$

but as x is not in the domain or codomain of σ, the substitution $\sigma \circ \{x \mapsto (V\{\sigma\})\}$ is equal to $\{x \mapsto V\} \circ \sigma$, so $C_1\{\sigma\}$ reduces to

$$\Theta_1\{\sigma\} \vdash \mathbf{let}\ B\{\{x \mapsto V\} \circ \sigma\}\ \mathbf{in}\ (E\{\{x \mapsto V\} \circ \sigma\}),$$

which is exactly $C_2\{\sigma\}$.

Evaluation context:
$\Psi ::= \Theta \vdash \Phi[\sigma]$
Restricted evaluation context:
$\phi ::= \Theta \vdash \Box[\sigma]$

Figure 17: Evaluation contexts in λ_{alloc}

□

Now, we prove that reduction by the CONTEXT rule is preserved inside any evaluation context.

Proposition 5 *If* $C_1 \overset{\text{CONTEXT}}{\longrightarrow} C_2$, *then for any context* Ψ, $\Psi[C_1] \overset{\text{CONTEXT}}{\longrightarrow} \Psi[C_2]$.

Proof Let $C_1 = \Theta_1 \vdash E_1$, $C_2 = \Theta_2 \vdash E_2$, $C_1' = \Psi[C_1]$, $C_2' = \Psi[C_2]$, and $\Psi = \Theta \vdash \Phi[\sigma]$. Let us assume w.l.o.g. that σ is correct for the considered objects. Then, $C_1' = (\Theta_1 + \Theta)\{\sigma\} \vdash \Phi[E_1]\{\sigma\}$ and $C_2' = (\Theta_2 + \Theta)\{\sigma\} \vdash \Phi[E_2]\{\sigma\}$.

Let us prove first that $C_1'' \overset{\text{CONTEXT}}{\longrightarrow} C_2''$, with $C_1'' = (\Theta_1 + \Theta) \vdash \Phi[E_1]$ and $C_2'' = (\Theta_2 + \Theta) \vdash \Phi[E_2]$. As we know, C_1 reduces to C_2 by rule CONTEXT, so in fact, $E_1 = \Phi[E_1']$, $E_2 = \Phi[E_2']$, and the proof of $C_1 \longrightarrow C_2$ is of the shape:

$$\frac{\Theta_1 \vdash E_1' \rightsquigarrow_c \Theta_2 \vdash E_2'}{C_1 \longrightarrow C_2}$$

But it is trivial that contraction rules are not affected by additional bindings in the heap, so we obtain easily that

$$\Theta + \Theta_1 \vdash E_1' \rightsquigarrow_c \Theta + \Theta_2 \vdash E_2'$$

Then, by rule CONTEXT, we have

$$C_1'' \longrightarrow C_2''.$$

Finally, by proposition 4, we deduce that

$$C_1''\{\sigma\} \longrightarrow C_2''\{\sigma\},$$

which is the expected result.

□

Now, we would like a similar property to be true with any reduction, but we have seen that it does not hold because of the toplevel nature of the LETREC rule. However, we have a slightly property, with contexts of the shape $\Theta \vdash \Box[\sigma]$, which we denote by the meta-variable ϕ, and call *weak evaluation contexts*. (The two notions of contexts introduced in this section are recalled in figure 17.) A toplevel reduction remains toplevel inside a weak evaluation context.

Proposition 6 *If* $C_1 \longrightarrow C_2$, *then* $\phi[C_1] \longrightarrow \phi[C_2]$.

5.2 Definition of the two translations

This section describes the translation. It consists in fact in two translations. The first one, called the *standard* translation, is very intuitive, but not easily proved correct. The second one is much less intuitive, but is easier to prove correct. The key technical point is that the standard translation reduces to the second translation, without using the BETA or PROJECT rules, and therefore without performing any real computation.

Both translations rely on a function **Size** from to $\lambda_\circ$ expressions to $\mathbb{N} \cup \{[?]\}$. This function is supposed to guess the size of the result 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, for any $e_\downarrow \in$ **Predictable**, **Size**$(e_\downarrow) \neq [?]$, and for any variable x, **Size**$(x) = [?]$.

Translation of expressions:

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![\lambda x.e]\!] & = & \lambda x.[\![e]\!] \\
[\![e_1 e_2]\!] & = & [\![e_1]\!][\![e_2]\!] \\
[\![\{\dots X_i = e_i \dots\}]\!] & = & \{\dots X_i = [\![e_i]\!] \dots\} \\
[\![e.X]\!] & = & [\![e]\!].X \\
[\![\textbf{let rec } b \textbf{ in } e]\!] & = & \textbf{let } \textbf{Dummy}(b), \textbf{Update}(b) \textbf{ in } [\![e]\!]
\end{array}
$$

Dummy pre-allocation of bindings:

$$
\begin{array}{lcll}
\textbf{Dummy}(\epsilon) & = & \epsilon & \\
\textbf{Dummy}(x = e, b) & = & (x = \textbf{alloc}\, n, \textbf{Dummy}(b)) & \text{if } \textbf{Size}(e) = n \\
\textbf{Dummy}(x = e, b) & = & \textbf{Dummy}(b) & \text{if } \textbf{Size}(e) = [?]
\end{array}
$$

Computation of bindings:

$$
\begin{array}{lcll}
\textbf{Update}(\epsilon) & = & \epsilon & \\
\textbf{Update}(x = e, b) & = & (y = (\textbf{update}\, x [\![e]\!]), \textbf{Update}(b)) & \text{if } \textbf{Size}(e) = n, \text{ with } y \text{ fresh} \\
\textbf{Update}(x = e, b) & = & (x = [\![e]\!], \textbf{Update}(b)) & \text{if } \textbf{Size}(e) = [?]
\end{array}
$$

Figure 18: Translation (standard translation)

The standard translation The standard translation is defined in figure 18. It is almost direct 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 of them is called the *pre-allocation* binding, and gives instructions to allocate dummy blocks on the heap for definitions of known size. The second binding is called the *update* binding. It computes definitions, and alternatively updates the previously pre-allocated dummy blocks for definitions of known sizes, or simply binds the result for definitions of unknown sizes. As announced, this translation does not map results to results. A simple example is $\lambda x.x$, which is translated as $\lambda x.x$. To reach a result, this translation still has to reduce to the configuration $\{l \mapsto (\lambda x.x)\} \vdash l$.

The second translation, named the *TOP* translation, performs all this kind of reductions at the meta-level, in order to associate results to results. As a consequence, it associates λ_{alloc} configurations to $\lambda_\circ$ expressions, and λ_{alloc} configurations to $\lambda_\circ$ configurations. It is defined in figures 19 and 20.

The TOP translation 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: $\{l \mapsto \lambda x.[\![e]\!]\} \vdash l$.

An evaluated record takes the translations of its fields and puts them in a record allocated on the heap at a fresh location l: $\Theta + \{l \mapsto \{S_v\}\} \vdash l$. Here, $\Theta \vdash S_v$ is the translation of the record s_v, defined in figure 19. If $s_v = (X_1 = v_1 \dots X_n = v_n)$, and for each i, $[\![v_i]\!]^{\text{TOP}} = \Theta_i \vdash V_i$, 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 $X = e, s$. s_v 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.

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 in translating e with the TOP translation, and then selecting the field X.

Translation of expressions as configurations:

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

Translation of configurations:

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

Translation of bindings and evaluated records:

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

Figure 19: The TOP translation (first part)

Translation of evaluated bindings: Ev. binding $\to$ (heap $\times$ substitution $\times$ variable allocation)

$$\begin{array}{lcl}
\mathbf{TOP}(\epsilon) & = & \emptyset \vdash (id, id) \\
\mathbf{TOP}(x = v, b_v) & = & \Theta \vdash (\sigma \circ \{x \mapsto V\}, \eta) \quad \text{if} \left\{ \begin{array}{l} \mathbf{Size}(v) = [?] \\ [\![v]\!]^{\mathrm{TOP}} = \emptyset \vdash V \\ \mathbf{TOP}(b_v) = \Theta \vdash (\sigma, \eta) \end{array} \right. \\
\mathbf{TOP}(x = v, b_v) & = & \Theta \vdash (\sigma, \eta \cup \{x \mapsto l\}) \quad \text{if} \left\{ \begin{array}{l} \mathbf{Size}(v) = n \\ [\![v]\!]^{\mathrm{TOP}} = \Theta \vdash l \\ \mathbf{TOP}(b_v) = \Theta \vdash (\sigma, \eta) \end{array} \right.
\end{array}$$

Actual dummy pre-allocation: Binding $\to$ (heap $\times$ variable allocation)

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

Actual computation of bindings: Binding $\to$ (heap $\times$ binding of λ_{alloc})

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

Figure 20: The TOP translation (continued): bindings

TOP translation of bindings The 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 rest of the binding b, is translated as follows. The pre-allocation pass, in the standard translation, 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, consists in either updating a dummy block with the translation of the definition, or simply binding it. Here, it is almost the same, except that the first definition is translated with the TOP translation, while the remaining ones are translated with the standard translation. The **TUp** is in charge of these operations.

Roughly, the binding b_v is translated as a heap and a substitution, by the **TOP** function. Definitions of unknown size $x = v$ yield a translation of the shape $\emptyset \vdash V$, and are included in the translation as a substitution $x \mapsto V$. Definitions of known size $x = v$ are translated as a heap and a variable allocation: v has a translation of the shape $\Theta \vdash l$, and it is included in the translation of b_v as Θ, and the allocation $x \mapsto l$.

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

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

5.3 Relating the two translations

An interesting fact is that the standard translation of any expression reduces to its TOP translation, in any context. The proof of this property is in three steps. First, we prove it for values. Then, we prove that the standard translation of a binding reduces to its TOP translation. Finally, we prove the expected result.

In fact, for values, we prove a more powerful result, namely that the standard translation reduces to the TOP translation, but only by rule Context, with a premise using Allocate, which we write Context (Allocate).

We make some additional hypotheses related to the correctness of the **Size** function.

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

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

Proposition 7 (Translation of values reduces to TOP) *For all context Ψ and for all value v, $\Psi[\emptyset \vdash [\![v]\!]] \longrightarrow^* \Psi[[\![v]\!]^{\mathrm{TOP}}]$, only by rule* Context *(*Allocate*).*

Proof By induction on v.

- $v = x$, trivial.
- $v = \lambda x.e$. Then $[\![v]\!] = \lambda x.[\![e]\!]$, so in any context $\emptyset \vdash [\![v]\!]$ reduces in one CONTEXT (ALLOCATE) step to $\{l \mapsto \lambda x.[\![e]\!]\} \vdash l$, which is the TOP translation of v.
- $v = \{X_1 = v_1 \dots X_n = v_n\}$. By induction hypothesis, for any context Ψ_i, for each i, we have

  $$\Psi_i[\emptyset \vdash [\![v_i]\!]] \longrightarrow \Psi_i[[\![v_i]\!]^{\mathrm{TOP}}].$$

  Let for each i, $[\![v_i]\!]^{\mathrm{TOP}} = \Theta_i \vdash V_i$. By a trivial induction on n, we prove that for any context Ψ,

  $$\Psi[\emptyset \vdash [\![\{X_1 = v_1 \dots X_n = v_n\}]\!]] \longrightarrow^* \Psi[\biguplus_{1 \le i \le n} \Theta_i \vdash \{X_1 = V_1 \dots X_n = V_n\}],$$

  only by rule CONTEXT (ALLOCATE). By proposition 5, this configuration in turn reduces by rule CONTEXT (ALLOCATE) to

  $$\Psi[\biguplus_{1 \le i \le n} \Theta_i + \{l \mapsto \{X_1 = V_1 \dots X_n = V_n\}\} \vdash l],$$

  which is exactly $[\![v]\!]^{\mathrm{TOP}}$.

□

Corollary 3 *For all weak evaluation context ϕ, expression E, and binding b of the shape $b = (x = v, b')$,*

$$\phi \circ \mathbf{Update}(b)[\emptyset \vdash E] \longrightarrow^* \phi \circ \mathbf{TUp}(b)[\emptyset \vdash E]$$

Proof We know that $\phi \circ \mathbf{Update}(b)[\emptyset \vdash E] = \phi[\mathbf{let}\ y = \Phi[[\![v]\!]], \mathbf{Update}(b')\ \mathbf{in}\ E]$,
where $(y, \Phi) = \begin{cases} (x, \Box) \text{ if } \mathbf{Size}(v) = [?] \\ (z, \mathbf{update}\, x\ \Box) \text{ otherwise } (z \text{ fresh}). \end{cases}$

This expression can be seen as $\Psi[\emptyset \vdash v]$ for some Ψ. By proposition 7, it reduces to $\Psi[[\![v]\!]^{\mathrm{TOP}}]$, so we obtain $\phi \circ \mathbf{let}\ y = \Phi, \mathbf{Update}(b')\ \mathbf{in}\ E[[\![v]\!]^{\mathrm{TOP}}]$, which is exactly $\phi \circ \mathbf{TUp}(b)[\emptyset \vdash E]$. □

Now, let us have a look at the translation of bindings. The TOP translation splits the bindings in two, according to the first non-value definition. But of course, one could split at another point, provided the first part contains only values. Indeed, the first part is given as an argument to the **TOP** function, which is defined only on evaluated bindings, whereas the second part is given as an argument to the **TDum** and **TUp** functions, which work as well on value and non-value definitions. We call a partial translation of a binding $b = b_v, b_v{}', b'$ its TOP translation, computed as if $b_v{}'$ was not evaluated, i.e. $\mathbf{TDum}(b_v{}', b') \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(b_v{}', b')$. We prove that any partial translation reduces to the TOP translation. We proceed in three main steps: first, we prove that the pre-allocation pass is performed at the object level by the code generated by the **Dummy** function, and at the meta level by the **TDum** function, in the same way ; then we prove a similar property for the functions **Update** and **TUp** ; and we eventually connect the two to prove the whole desired property.

Proposition 8 (Dummy) *For all binding B, for all weak evaluation context ϕ,*

$$\phi[\emptyset \vdash \mathbf{let}\ \ \mathbf{Dummy}(b), B\ \mathbf{in}\ E] \longrightarrow^* (\phi \circ \mathbf{TDum}(b))[\emptyset \vdash \mathbf{let}\ B\ \mathbf{in}\ E].$$

Proof By induction on b. If b is empty, then there is nothing to prove. Otherwise, we are in one of the following cases.

- $b = (x = e, b')$, with $\mathbf{Size}(e) = [?]$. Then $\mathbf{Dummy}(b) = \mathbf{Dummy}(b')$ and $\mathbf{TDum}(b) = \mathbf{TDum}(b')$, so by induction hypothesis, we obtain the expected result.
- $b = (x = e, b')$, with $\mathbf{Size}(e) = n$. Then $\mathbf{Dummy}(b) = (x = \mathbf{alloc}\, n, \mathbf{Dummy}(b'))$. Let $\mathbf{TDum}(b') = \Theta \vdash \eta$, we have $\mathbf{TDum}(b) = \Theta + \{l \mapsto \mathbf{alloc}\, n\} \vdash \eta \cup \{x \mapsto l\}$, for a fresh l. Let ϕ be a weak evaluation context, and $E_0 = \phi[\mathbf{let}\ \mathbf{Dummy}(b), B\ \mathbf{in}\ E]$. We have $E_0 = \phi[\emptyset \vdash \mathbf{let}\ x = \mathbf{alloc}\, n, \mathbf{Dummy}(b'), B\ \mathbf{in}\ E]$. By rule CONTEXT (ALLOCATE), we have $E_0 \longrightarrow \phi[\{l \mapsto \mathbf{alloc}\, n\} \vdash \mathbf{let}\ x = l, \mathbf{Dummy}(b'), B\ \mathbf{in}\ E]$. By proposition 6, this last expression reduces to $\phi[\{l \mapsto \mathbf{alloc}\, n\} \vdash (\mathbf{let}\ \mathbf{Dummy}(b'), B\ \mathbf{in}\ E)\{x \mapsto l\}]$. Let $\phi_0 = \mathbf{TDum}(x = e) = \{l \mapsto \mathbf{alloc}\, n\} \vdash \Box[\{x \mapsto l\}]$ and $\phi_1 = \phi \circ \phi_0$; we can view the expression as $\phi_1[\emptyset \vdash \mathbf{let}\ \mathbf{Dummy}(b'), B\ \mathbf{in}\ E]$, which by induction hypothesis reduces to $\phi_1[\mathbf{TDum}(b')[\emptyset \vdash \mathbf{let}\ B\ \mathbf{in}\ E]]$. In other words, we obtain $\phi[\mathbf{TDum}(x = e) \circ \mathbf{TDum}(b')[\emptyset \vdash \mathbf{let}\ B\ \mathbf{in}\ E]]$, which is the expected result, since obviously $\mathbf{TDum}(x = e) \circ \mathbf{TDum}(b') = \mathbf{TDum}(b)$.

□

Proposition 9 (Update) *Let $b = (x = v, b')$. For all weak evaluation context ϕ, for all expression E, we have*

$$\phi \circ \mathbf{TDum}(b) \circ \mathbf{TUp}(b)[\emptyset \vdash E] \longrightarrow^* \phi \circ \mathbf{TDum}(b') \circ \mathbf{TOP}(x = v) \circ \mathbf{Update}(b')[\emptyset \vdash E].$$

Proof

- If $\mathbf{Size}(v) = n$, then $[\![v]\!]^{\mathrm{TOP}} = \Theta_v \vdash l$, and we have

  $$\mathbf{TUp}(b) = \Theta_v \vdash y = \mathbf{update}\, x\ l, \mathbf{Update}(b'),$$

  with a fresh y. Alternatively, we can choose another fresh location l' for the result, and have $[\![v]\!]^{\mathrm{TOP}} = \Theta'_v \vdash l'$, with $\Theta'_v = \Theta_{v \setminus l} + \{l' \mapsto \Theta_v(l)\}$.

  Let $E_0 = \phi \circ \mathbf{TDum}(b) \circ \mathbf{TUp}(b)[\emptyset \vdash E]$.

  We have $E_0 = \phi \circ \mathbf{TDum}(b)[\Theta'_v \vdash \mathbf{let}\ \ y = \mathbf{update}\, x\ l', \mathbf{Update}(b')\ \ \mathbf{in}\ \ E]$, and also $\mathbf{Size}(v) = n$ and $\mathbf{TDum}(b) = \mathbf{TDum}(b') \circ (\{l \mapsto \mathbf{alloc}\, n\} \vdash \{x \mapsto l\})$. So

  $$E_0 = \phi \circ \mathbf{TDum}(b')[(\Theta'_v + \{l \mapsto \mathbf{alloc}\, n\} \vdash \mathbf{let}\ \ y = \mathbf{update}\, x\ l', \mathbf{Update}(b')\ \ \mathbf{in}\ \ E)\{x \mapsto l\}].$$

  But by hypothesis 1, $\mathbf{Size}(\Theta'_v(l')) = n$, so rule UPDATE applies, and E_0 reduces to

  $$\phi \circ \mathbf{TDum}(b')[(\Theta'_v + \{l \mapsto \Theta'_v(l')\} \vdash \mathbf{let}\ \ y = \{\}, \mathbf{Update}(b')\ \ \mathbf{in}\ \ E)\{x \mapsto l\}],$$

  and then, as y is fresh, by rule LET to

  $$\phi \circ \mathbf{TDum}(b')[(\Theta'_v + \{l \mapsto \Theta'_v(l')\} \vdash \mathbf{let}\ \ \mathbf{Update}(b')\ \ \mathbf{in}\ \ E)\{x \mapsto l\}].$$

  But the location l' is not used anymore, so by rule GC, the obtained expression reduces to

  $$\phi \circ \mathbf{TDum}(b')[(\Theta'_{v \setminus l'} + \{l \mapsto \Theta'_v(l')\} \vdash \mathbf{let}\ \ \mathbf{Update}(b')\ \ \mathbf{in}\ \ E)\{x \mapsto l\}].$$

  And finally, we notice that $\Theta'_{v \setminus l'} + \{l \mapsto \Theta'_v(l')\} = \Theta_v$, so E_0 reduces to

  $$\begin{array}{l} \phi \circ \mathbf{TDum}(b')[(\Theta_v \vdash \mathbf{let}\ \ \mathbf{Update}(b')\ \ \mathbf{in}\ \ E)\{x \mapsto l\}] \\ = \phi \circ \mathbf{TDum}(b') \circ \mathbf{TOP}(x = v)[\emptyset \vdash \mathbf{let}\ \ \mathbf{Update}(b')\ \ \mathbf{in}\ \ E] \\ = \phi \circ \mathbf{TDum}(b') \circ \mathbf{TOP}(x = v) \circ \mathbf{Update}(b')[\emptyset \vdash E]. \end{array}$$

- If $\mathbf{Size}(v) = [?]$, then there exists a y such that $[\![v]\!]^{\mathrm{TOP}} = \emptyset \vdash y$, so

  $$\mathbf{TUp}(b) = \emptyset \vdash x = y, \mathbf{Update}(b').$$

  Let $E_0 = \phi \circ \mathbf{TDum}(b) \circ \mathbf{TUp}(b)[\emptyset \vdash E]$.

  We have $E_0 = \phi \circ \mathbf{TDum}(b)[\emptyset \vdash \mathbf{let}\ \ x = y, \mathbf{Update}(b')\ \ \mathbf{in}\ \ E]$,

  and by rule LET, by proposition 6, $E_0 \longrightarrow \phi \circ \mathbf{TDum}(b)[\emptyset \vdash (\mathbf{let}\ \ \mathbf{Update}(b')\ \ \mathbf{in}\ \ E)\{x \mapsto y\}]$.

  But $\mathbf{TOP}(x = v) = \mathbf{TOP}(x = y) = \emptyset \vdash (x \mapsto y, id)$, so $E_0 \longrightarrow \phi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(x = v)[\emptyset \vdash \mathbf{let}\ \ \mathbf{Update}(b')\ \ \mathbf{in}\ \ E]$, which is the expected result.

□

Proposition 10 (Pre-allocated locations are definitive) *If $\mathbf{TDum}(b_v) = \Theta_1 \vdash \eta_1$, then there exist $\Theta_2, \sigma_2, \eta_2$ such that $\mathbf{TOP}(b_v) = \Theta_2 \vdash (\sigma_2, \eta_2)$ and $\eta_1 = \eta_2$.*

In the following proposition, we consider a substitution σ as a context $\emptyset \vdash \Box[\sigma]$.

Proposition 11 (Decomposition of the translation of evaluated bindings) *Let $b_v = (x = v, {b_v}')$ and $\mathbf{TDum}({b_v}') = \Theta_{{b_v}'} \vdash \eta_{{b_v}'}$. We have*

$$\mathbf{TOP}(b_v) = \eta_{{b_v}'} \circ \mathbf{TOP}(x = v) \circ \mathbf{TOP}({b_v}').$$

Proof Let $\mathbf{TOP}(x = v) = \Theta_v \vdash (\sigma_v, \eta_v)$, and $\mathbf{TOP}(b_v{}') = \Theta \vdash (\sigma, \eta)$. We have $\mathbf{TOP}(b_v) = \Theta_v + \Theta_{b_v{}'} \vdash (\sigma \circ \sigma_v, \eta \cup \eta_v)$. By proposition 10, we can choose Θ, σ, and η such that $\eta = \eta_{b_v{}'}$. Then,

$$\begin{aligned}&\eta_{b_v{}'} \circ \mathbf{TOP}(x = v) \circ \mathbf{TOP}(b_v{}')\\ &= \Theta + \Theta_v \vdash \sigma \circ \eta \circ \sigma_v \circ \eta_v \circ \eta_{b_v{}'}\\ &= \Theta + \Theta_v \vdash \sigma \circ \eta_{b_v{}'} \circ \sigma_v \circ \eta_v \circ \eta_{b_v{}'}\end{aligned}$$

But η_v and $\eta_{b_v{}'}$ have disjoint domains and codomains, so they commute and we obtain

$$\begin{aligned}&\eta_{b_v{}'} \circ \mathbf{TOP}(x = v) \circ \mathbf{TOP}(b_v{}')\\ &= \Theta + \Theta_v \vdash \sigma \circ \eta_{b_v{}'} \circ \sigma_v \circ \eta_{b_v{}'} \circ \eta_v\end{aligned}$$

Furthermore, $\eta_{b_v{}'}$ and σ_v also have disjoint domains and codomains, so they commute. Finally, $\eta_{b_v{}'}$ is idempotent, so

$$\begin{aligned}&\eta_{b_v{}'} \circ \mathbf{TOP}(x = v) \circ \mathbf{TOP}(b_v{}')\\ &= \Theta + \Theta_v \vdash \sigma \circ \sigma_v \circ \eta_{b_v{}'} \circ \eta_v\\ &= \Theta + \Theta_v \vdash (\sigma \circ \sigma_v) \circ (\eta_{b_v{}'} \cup \eta_v)\\ &= \mathbf{TOP}(b_v)\end{aligned}$$

□

Proposition 12 (Commuting contexts) *Let* $\phi_1 = \Theta_1 \vdash \Box[\sigma_1]$ *and* $\phi_2 = \Theta_2 \vdash \Box[\sigma_2]$. *If* $\mathbf{dom}(\sigma_2) \perp \sigma_1$ *and* $\sigma_1^2 = \sigma_1$, *then* $\phi_1 \circ \phi_2 = \sigma_1 \circ \phi_2 \circ \phi_1$.

Proof This property is simple, provided $\sigma_2 \circ \sigma_1 = \sigma_1 \circ \sigma_2 \circ \sigma_1$. Recall that $\mathbf{dom}(\sigma_2) \perp \sigma_1$. We prove that the two total functions $\sigma = \sigma_2 \circ \sigma_1$ and $\sigma' = \sigma_1 \circ \sigma_2 \circ \sigma_1$ from variables to values are pointwise equal.

- On $x \in \mathbf{dom}(\sigma_2)$, by hypothesis $x \notin \mathbf{dom}(\sigma_1)$, so we have $\sigma'(x) = x\{\sigma_1\}\{\sigma_2\}\{\sigma_1\} = x\{\sigma_2\}\{\sigma_1\} = \sigma(x)$.
- On $x \notin \mathbf{dom}(\sigma_2)$, distinguish the two cases.
  - If $x \in \mathbf{dom}(\sigma_1)$, then $\sigma(x) = x\{\sigma_2\}\{\sigma_1\} = x\{\sigma_1\}$. But by hypothesis $\sigma_1(x) \in \mathbf{cod}(\sigma_1) \perp \mathbf{dom}(\sigma_2)$, so $\sigma'(x) = x\{\sigma_1\}\{\sigma_2\}\{\sigma_1\} = \sigma_1(x)\{\sigma_2\}\{\sigma_1\} = \sigma_1(x)\{\sigma_1\} = x\{\sigma_1^2\} = x\{\sigma_1\} = \sigma(x)$.
  - If $x \notin \mathbf{dom}(\sigma_1)$, then $\sigma(x) = x = \sigma'(x)$.

□

Corollary 4 *Let* $b_v = (b_{v1}, b_{v2})$ *be a syntactically correct binding. Let* $\mathbf{TDum}(b_{v2}) = \Theta_2 \vdash \eta_2$. *We have* $\eta_2 \circ \mathbf{TOP}(b_{v1}) \circ \eta_2 = \eta_2 \circ \mathbf{TOP}(b_{v1})$.

Proof Let $\mathbf{TOP}(b_{v1}) = \Theta_1 \vdash (\sigma_1, \eta_1)$. By proposition 12, it is enough to prove $\mathbf{dom}(\sigma_1 \circ \eta_1) \perp \eta_2$ and $\eta_2^2 = \eta_2$. But we have $\mathbf{dom}(b_{v1}) \perp \mathbf{dom}(b_{v2})$, so $\mathbf{dom}(\sigma_1 \circ \eta_1) \perp \mathbf{dom}(\eta_2)$. Moreover, $\mathbf{cod}(\eta_2)$ contains only locations, whereas $\mathbf{dom}(\sigma_1 \circ \eta_1)$ contains only variables, so $\mathbf{cod}(\sigma_2) \perp \mathbf{dom}(\sigma_1 \circ \eta_1)$. Finally, as all variable allocations, η_2 is idempotent. □

Corollary 5 *Let* $b_v = (b_{v1}, b_{v2})$ *and* $\mathbf{TDum}(b_{v2}) = \Theta_2 \vdash \eta_2$. *We have*

$$\mathbf{TOP}(b_v) = \eta_2 \circ \mathbf{TOP}(b_{v1}) \circ \mathbf{TOP}(b_{v2}).$$

Proof By induction on b_{v1}.

- $b_{v1} = \epsilon$, because η_2 is idempotent.

- $b_{v1} = (x = v, b_{v1}')$. Let $b_v{}' = b_{v1}', b_{v2}$, $\mathbf{TDum}(b_v{}') = \Theta'_{b_v{}'} \vdash \eta_{b_v{}'}$, and $\mathbf{TDum}(b_{v1}') = \Theta'_{b_{v1}'} \vdash \eta_{b_{v1}'}$. By definition of **TDum**, we have $\eta_{b_v{}'} = \eta_{b_{v1}'} \cup \eta_2$. Then, we can calculate

$$\begin{aligned}
\mathbf{TOP}(b_v) &= \eta_{b_v{}'} \circ \mathbf{TOP}(x = v) \circ \mathbf{TOP}(b_v{}') \\
&\qquad\qquad \text{(by lemma 11)} \\
&= \eta_{b_v{}'} \circ \mathbf{TOP}(x = v) \circ \eta_2 \circ \mathbf{TOP}(b_{v1}') \circ \mathbf{TOP}(b_{v2}) \\
&\qquad\qquad \text{(by induction hypothesis)} \\
&= \eta_{b_{v1}'} \cup \eta_2 \circ \mathbf{TOP}(x = v) \circ \eta_2 \circ \mathbf{TOP}(b_{v1}') \circ \mathbf{TOP}(b_{v2}) \\
&= \eta_{b_{v1}'} \circ \underline{\eta_2 \circ \mathbf{TOP}(x = v) \circ \eta_2} \circ \mathbf{TOP}(b_{v1}') \circ \mathbf{TOP}(b_{v2}) \\
&= \eta_{b_{v1}'} \circ \eta_2 \circ \overline{\mathbf{TOP}(x = v) \circ \mathbf{TOP}(b_{v1}')} \circ \mathbf{TOP}(b_{v2}) \\
&\qquad\qquad \text{(by proposition 4)} \\
&= \eta_2 \circ \underline{\eta_{b_{v1}'} \circ \mathbf{TOP}(x = v) \circ \mathbf{TOP}(b_{v1}')} \circ \mathbf{TOP}(b_{v2}) \\
&= \eta_2 \circ \overline{\mathbf{TOP}(b_{v1}) \circ \mathbf{TOP}(b_{v2})} \\
&\qquad\qquad \text{(by proposition 11)}
\end{aligned}$$

□

Proposition 13 (TOP Update pass) *For all weak evaluation context ϕ, and configuration C,*

$$\phi \circ \mathbf{TDum}(b_v, b) \circ \mathbf{TUp}(b_v, b)[C] \longrightarrow^* \phi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b)[C].$$

Proof By induction on b_v. If $b_v = \epsilon$, there is nothing to prove. Otherwise, let $b_v = (x = v, b_v{}')$. By proposition 9,

$$\phi \circ \mathbf{TDum}(b_v, b) \circ \mathbf{TUp}(b_v, b)[C] \longrightarrow^* \phi \circ \mathbf{TDum}(b_v{}', b) \circ \mathbf{TOP}(x = v) \circ \mathbf{Update}(b_v{}', b)[C].$$

But by corollary 4, this is equal to

$$\phi \circ \eta \circ \mathbf{TOP}(x = v) \circ \mathbf{TDum}(b_v{}', b) \circ \mathbf{Update}(b_v{}', b)[C],$$

where $\mathbf{TDum}(b_v{}', b) = \Theta \vdash \eta$.

By induction hypothesis, we know that the obtained expression reduces to

$$\phi \circ \eta \circ \mathbf{TOP}(x = v) \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v{}') \circ \mathbf{Update}(b)[C].$$

But if we let $\mathbf{TDum}(b_v{}') = \Theta_1 \vdash \eta_1$ and $\mathbf{TDum}(b) = \Theta_2 \vdash \eta_2$, we have $\eta = \eta_1 \cup \eta_2$, so

$$\begin{aligned}
&\phi \circ \eta \circ \mathbf{TOP}(x = v) \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v{}') \circ \mathbf{Update}(b)[C] \\
&= \phi \circ \eta_1 \circ \underline{\eta_2 \circ \mathbf{TOP}(x = v) \circ \mathbf{TDum}(b)} \circ \mathbf{TOP}(b_v{}') \circ \mathbf{Update}(b)[C] \\
&= \phi \circ \eta_1 \circ \overline{\mathbf{TDum}(b) \circ \mathbf{TOP}(x = v)} \circ \mathbf{TOP}(b_v{}') \circ \mathbf{Update}(b)[C] \\
&\qquad\qquad (\text{ by corollary 4 }) \\
&= \phi \circ \mathbf{TDum}(b) \circ \underline{\eta_1 \circ \mathbf{TOP}(x = v) \circ \mathbf{TOP}(b_v{}')} \circ \mathbf{Update}(b)[C] \\
&\qquad (\text{ because } \mathbf{TDum}(b) \text{ is not modified by any substitution }) \\
&\phi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b)[C] \\
&\qquad\qquad (\text{ by proposition 11 })
\end{aligned}$$

□

Proposition 14 (Update pass) *For all weak evaluation context ϕ, and configuration C,*

$$\phi \circ \mathbf{TDum}(b_v, b) \circ \mathbf{Update}(b_v, b)[C] \longrightarrow^* \phi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b)[C].$$

Proof By corollary 3, we have

$$\begin{aligned}
&\phi \circ \mathbf{TDum}(b_v, b) \circ \mathbf{Update}(b_v, b)[C] \\
&\longrightarrow^* \phi \circ \mathbf{TDum}(b_v, b) \circ \mathbf{TUp}(b_v, b)[C].
\end{aligned}$$

By proposition 13, it further reduces to $\phi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b)[C]$. □

Proposition 15 (Partial translation of bindings) *For all evaluation context Ψ,*

$$\Psi[\emptyset \vdash [\![\mathbf{let\ rec}\ \ b_v, b\ \ \mathbf{in}\ \ e]\!]] \longrightarrow^* \Psi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b)[\emptyset \vdash e].$$

Proof Let $\Psi = \Theta \vdash \Phi[\sigma]$, and $\phi = \Theta \vdash \Box[\sigma]$. Let

$$E_0 = \Psi[\emptyset \vdash [\![e]\!]] = \Psi[\emptyset \vdash \textbf{let } \mathbf{Dummy}(b_v, b), \mathbf{Update}(b_v, b) \textbf{ in } [\![e]\!]]$$

By rule LIFT and modulo variable renaming, we have

$$E_0 \longrightarrow^* \phi[\emptyset \vdash \textbf{let } \mathbf{Dummy}(b_v, b), \mathbf{Update}(b_v, b) \textbf{ in } \Phi[[\![e]\!]]].$$

By proposition 8, this expression reduces to $\phi \circ \mathbf{TDum}(b_v, b)[\emptyset \vdash \textbf{let } \mathbf{Update}(b_v, b) \textbf{ in } \Phi[[\![e]\!]]]$.

By proposition 14, it in turn reduces to $\phi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b)[\Phi[[\![e]\!]]]$, which is equal to $\Psi \circ \mathbf{TDum}(b) \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b)[[\![e]\!]]$. □

Lemma 1 (Standard translation reduces to TOP translation) *For all context Ψ and for all expression e,*

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

Proof By induction on e. If e is a value, we use proposition 7.

Application. Let $e = e_1 e_2$, Ψ be a context, and $E_0 = \Psi[\emptyset \vdash [\![e]\!]] = \Psi[\emptyset \vdash [\![e_1]\!][\![e_2]\!]]$. Let also $[\![e_1]\!]^{\mathrm{TOP}} = \Theta_1 \vdash E_1$. By induction hypothesis, $E_0 \longrightarrow^* \Psi[\Theta_1 \vdash E_1[\![e_2]\!]]$. If e_1 is not a value, this is directly $\Psi[[\![e]\!]^{\mathrm{TOP}}]$. Otherwise, E_1 is a value, say V_1, and $\Psi_0 = \Psi[\Theta_1 \vdash (V_1 \Box)[id]]$ is an evaluation context, so by induction hypothesis again, if we let $[\![e_2]\!]^{\mathrm{TOP}} = \Theta_2 \vdash E_2$, then $\Psi_0[\emptyset \vdash [\![e_2]\!]] \longrightarrow^* \Psi_0[\Theta_2 \vdash E_2]$, which is equal to $\Psi[\Theta_1 + \Theta_2 \vdash V_1 E_2] = \Psi[[\![e]\!]^{\mathrm{TOP}}]$.

Record field selection. Simple by induction hypothesis.

Record. Let $e = \{s_v, X = f, s\}$, where f is not a value. Let $[\![s_v]\!]^{\mathrm{TOP}} = \Theta_1 \vdash S_v$. By a trivial induction on s_v, we prove that $\Psi[\emptyset \vdash [\![\{s_v, X = f, s\}]\!]] \longrightarrow^* \Psi[\Theta_1 \vdash \{S_v, X = [\![f]\!], [\![s]\!]\}]$. This expression can be viewed as $\Psi_0[\emptyset \vdash [\![f]\!]]$, with $\Psi_0 = \Psi[\Theta_1 \vdash \{S_v, X = \Box, [\![s]\!]\}]$. Let $[\![f]\!]^{\mathrm{TOP}} = \Theta_2 \vdash F$. By induction hypothesis, the above expression reduces to $\Psi_0[\Theta_2 \vdash F]$, which is equal to $\Psi[\Theta_1 + \Theta_2 \vdash \{S_v, X = F, [\![s]\!]\}]$, and this is the expected result.

Binding. Let $e = \textbf{let rec } b \textbf{ in } f$.

1. If $b = \epsilon$, then $[\![b]\!]^{\mathrm{TOP}} = \emptyset \vdash \Box[id]$, so $[\![e]\!]^{\mathrm{TOP}} = [\![f]\!]^{\mathrm{TOP}}$. So, $\Psi[\emptyset \vdash [\![e]\!]] = \Psi[\emptyset \vdash \textbf{let } \epsilon \textbf{ in } [\![f]\!]]$. By rules LIFT and then EMPTYLET, it reduces to $\Psi[\emptyset \vdash [\![f]\!]]$, which by induction hypothesis reduces to $\Psi[[\![f]\!]^{\mathrm{TOP}}]$, as expected.
2. If $b = b_v$, non empty, then $[\![e]\!]^{\mathrm{TOP}} = \mathbf{TOP}(b_v)[[\![f]\!]^{\mathrm{TOP}}]$. We have

$$\begin{aligned}
&\Psi[\emptyset \vdash [\![e]\!]] \\
&= \Psi[\emptyset \vdash [\![\textbf{let rec } b_v \textbf{ in } f]\!]] \\
&\longrightarrow^* \Psi \circ \mathbf{TOP}(b_v)[\emptyset \vdash [\![f]\!]] \\
&\qquad \text{(by proposition 15)} \\
&\longrightarrow^* \Psi \circ \mathbf{TOP}(b_v)[[\![f]\!]^{\mathrm{TOP}}] \\
&\quad \text{(by induction hypothesis)} \\
&= \Psi[[\![e]\!]^{\mathrm{TOP}}]
\end{aligned}$$

3. If $b = b_v, b'$, with b' non empty, then $[\![e]\!]^{\mathrm{TOP}} = \mathbf{TDum}(b') \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(b')[\emptyset \vdash [\![f]\!]]$. We have

$$\begin{aligned}
&\Psi[\emptyset \vdash [\![e]\!]] \\
&= \Psi[\emptyset \vdash [\![\textbf{let rec } b_v, b' \textbf{ in } f]\!]] \\
&\longrightarrow^* \Psi \circ \mathbf{TDum}(b') \circ \mathbf{TOP}(b_v) \circ \mathbf{Update}(b')[\emptyset \vdash [\![f]\!]] \\
&\qquad \text{(by proposition 15)} \\
&\longrightarrow^* \Psi \circ \mathbf{TDum}(b') \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(b')[\emptyset \vdash [\![f]\!]] \\
&\qquad \text{(by induction hypothesis)} \\
&= \Psi[[\![e]\!]^{\mathrm{TOP}}]
\end{aligned}$$

□

6 Correctness

6.1 Translation of contexts and compositionality

Both the standard and the TOP translations rely on sizes. In a binding, if a definition $x = e$ is of known size, then it is translated as the binding $y = \mathbf{update}\, x\, [\![e]\!]$, whereas otherwise, it is translated as $x = [\![e]\!]$. For this reason, it is not compositional in the usual sense: a straightforward property such as $[\![\mathbb{E}\,[e]]\!] = [\![\mathbb{E}\,]\!][\![e]\!]$ does not hold. Moreover, there is no straightforward translation for contexts: consider **let rec** $x = \Box$ **in** $\{\}$ for instance; should it be translated as if the expression filling the hole was of known size or unknown size?

The TOP translation retains a kind of compositionality though. We define *complete contexts* in $\lambda_\circ$, as normal contexts, except that the context hole is now annotated with a size indication $\zeta \in \mathbb{N} \cup \{[?]\}$. Complete context application is only valid if the argument as the expected size. Complete contexts are then translated exactly as expressions. For this, the definition in figure 19 is simply extended with $[\![\Box_\zeta]\!]^{\mathrm{TOP}} = [\![\Box_\zeta]\!] = \Box$, given that a context hole $\Box_\zeta$ has size ζ, and that it is not a value. Normal contexts are translated, with an additional argument giving the size of the context hole. For instance, we write $[\![\mathbb{E}\,]\!]^{\mathrm{TOP}}_\zeta$ for $[\![\mathbb{E}\,[\Box_\zeta]]\!]^{\mathrm{TOP}}$. The standard translation is compositional for this notion of contexts.

Proposition 16 (Compositionality of the standard translation) *For all context* $\mathbb{E}$ *and expression* e*,*

$$[\![\mathbb{E}\,[e]]\!] = [\![\mathbb{E}\,]\!]_{\mathbf{Size}(e)}[\![e]\!].$$

The translation is compositional with respect to this notion of contexts, provided the right size indication is chosen, and that the expression filling the hole is not a value. Indeed, in the translation of bindings, a distinction is made between evaluated and unevaluated definitions, which breaks compositionality in this case, because the context hole is not considered a value. Fortunately, for values, a weaker property of compositionality modulo reduction holds, which allows to prove that the translation is faithfull.

Proposition 17 (Compositionality for lift contexts) *If* $e \notin$ **Values**, *then*

$$[\![\mathbb{L}\,[e]]\!]^{\mathrm{TOP}} = [\![\mathbb{L}\,]\!]^{\mathrm{TOP}}_{\mathbf{Size}(e)}[\![e]\!]^{\mathrm{TOP}}].$$

Proof By case analysis on $\mathbb{L}$. We treat one example case, application: $\mathbb{L} = \Box f$. We have $[\![\mathbb{L}\,[e]]\!]^{\mathrm{TOP}} = [\![ef]\!]^{\mathrm{TOP}} = \Theta \vdash E[\![f]\!]$, where $[\![e]\!]^{\mathrm{TOP}} = \Theta \vdash E$. But $[\![\mathbb{L}\,]\!]^{\mathrm{TOP}}_{\mathbf{Size}(e)} = \emptyset \vdash \Box[\![f]\!]$, which is the expected result. □

Proposition 18 (Compositionality for multiple lift contexts) *If* $e \notin$ **Values**, *then*

$$[\![\mathbb{F}\,[e]]\!]^{\mathrm{TOP}} = [\![\mathbb{F}\,]\!]^{\mathrm{TOP}}_{\mathbf{Size}(e)}[\![e]\!]^{\mathrm{TOP}}].$$

Proof By induction on $\mathbb{F}$. If $\mathbb{F} = \Box$, there is nothing to prove. Otherwise, let $\mathbb{F} = \mathbb{L}\,[\mathbb{F}']$ and $\zeta = \mathbf{Size}(e)$.
By induction hypothesis, $[\![\mathbb{F}'[e]]\!]^{\mathrm{TOP}} = [\![\mathbb{F}'\,]\!]^{\mathrm{TOP}}_\zeta[\![e]\!]^{\mathrm{TOP}}]$.
As the **Size** function is compositional, $\zeta' = \mathbf{Size}(\mathbb{F}'[e]) = \mathbf{Size}(\mathbb{F}'[\Box_\zeta])$.
By proposition 18, $[\![\mathbb{L}\,[\mathbb{F}'[e]]]\!]^{\mathrm{TOP}} = [\![\mathbb{L}\,]\!]^{\mathrm{TOP}}_{\zeta'}[\![\mathbb{F}'[e]]\!]^{\mathrm{TOP}}] = [\![\mathbb{L}\,]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}'\,]\!]^{\mathrm{TOP}}_\zeta[\![e]\!]^{\mathrm{TOP}}]]$.
By proposition 18, $[\![\mathbb{L}\,[\mathbb{F}'\,]]\!]^{\mathrm{TOP}}_\zeta = [\![\mathbb{L}\,[\mathbb{F}'[\Box_\zeta]]]\!]^{\mathrm{TOP}} = [\![\mathbb{L}\,]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}'[\Box_\zeta]]\!]^{\mathrm{TOP}}] = [\![\mathbb{L}\,]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}'\,]\!]^{\mathrm{TOP}}_\zeta]$.
So, $[\![\mathbb{L}\,[\mathbb{F}'[e]]]\!]^{\mathrm{TOP}} = [\![\mathbb{L}\,[\mathbb{F}'\,]]\!]^{\mathrm{TOP}}_\zeta[\![e]\!]^{\mathrm{TOP}}]$. □

Lemma 2 (Compositionality for evaluation contexts) *If* $e \notin$ **Values**, *then*

$$[\![\mathbb{E}\,[e]]\!]^{\mathrm{TOP}} = [\![\mathbb{E}\,]\!]^{\mathrm{TOP}}_{\mathbf{Size}(e)}[\![e]\!]^{\mathrm{TOP}}].$$

Proof By case on $\mathbb{E}$. Let $\zeta = \mathbf{Size}(e)$.

- If $\mathbb{E} = \mathbb{F}$, use proposition 18.
- If $\mathbb{E} = b_v \vdash \mathbb{F}$, then

$$\begin{aligned}
[\![\mathbb{E}\,[e]]\!]^{\mathrm{TOP}} &= \mathbf{TOP}(b_v)[[\![\mathbb{F}\,[e]]\!]^{\mathrm{TOP}}] \\
&= \mathbf{TOP}(b_v)[[\![\mathbb{F}\,]\!]^{\mathrm{TOP}}_\zeta[\![e]\!]^{\mathrm{TOP}}]] \\
&= (\mathbf{TOP}(b_v) \circ [\![\mathbb{F}\,]\!]^{\mathrm{TOP}}_\zeta)[\![e]\!]^{\mathrm{TOP}}] \\
&= [\![\mathbb{E}\,]\!]^{\mathrm{TOP}}_\zeta[\![e]\!]^{\mathrm{TOP}}].
\end{aligned}$$

- If $\mathbb{E} = (b_v, x = \mathbb{F}, b \vdash f)$, then let $b_0 = (x = \mathbb{F}[e], b)$. We have $[\![\mathbb{E}[e]]\!]^{\mathrm{TOP}} = \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(b_0)[\emptyset \vdash [\![f]\!]]$, since $\mathbb{F}[e]$ cannot be a value.

  Let $\Theta' \vdash E' = [\![\mathbb{F}[e]]\!]^{\mathrm{TOP}} = [\![\mathbb{F}]\!]^{\mathrm{TOP}}_{\zeta}[[\![e]\!]^{\mathrm{TOP}}]$ (by proposition 18).

  Let $\Theta_u \vdash B = \mathbf{TUp}(b)$, and $\zeta' = \mathbf{Size}(\mathbb{F}[e]) = \mathbf{Size}(\mathbb{F}[\Box_\zeta])$.

  Let $(x', \Phi') = \begin{cases} (x, E') \text{ if } \zeta' = [?] \\ (x, \mathbf{update}\, x\ E') \text{ otherwise} \end{cases}$

  We have $\mathbf{TUp}(b_0) = \Theta_u + \Theta' \vdash x' = \Phi'[E'], B$. Let $\Psi_0 = \Theta_u \vdash \mathbf{let}\ x' = \Phi', B\ \mathbf{in}\ [\![f]\!]$. We have

$$\begin{aligned}[\![\mathbb{E}[e]]\!]^{\mathrm{TOP}} &= \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ \Psi_0 \circ [\![\mathbb{F}]\!]^{\mathrm{TOP}}_{\zeta}[[\![e]\!]^{\mathrm{TOP}}] \\ &= [\![\mathbb{E}]\!]^{\mathrm{TOP}}_{\zeta}[[\![e]\!]^{\mathrm{TOP}}].\end{aligned}$$

□

When the expression filling the context hole is a value, we have seen that this compositionality property is false. We nevertheless prove a weaker one.

Proposition 19 (Semi-compositionality for lift contexts) *For all evaluation context* Ψ*,*

$$\Psi[[\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\mathbf{Size}(v)}[[\![v]\!]^{\mathrm{TOP}}]] \longrightarrow^* \Psi[[\![\mathbb{L}[v]]\!]^{\mathrm{TOP}}].$$

Proof By case on $\mathbb{L}$. Let $\zeta = \mathbf{Size}(v)$ and $\Theta_v \vdash V = [\![v]\!]^{\mathrm{TOP}}$.

- If $\mathbb{L}$ is of the shape $v'\Box$ or $\Box.X$, then $\Psi[[\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\mathbf{Size}(v)}[[\![v]\!]^{\mathrm{TOP}}]] = \Psi[[\![\mathbb{L}[v]]\!]^{\mathrm{TOP}}]$.
- $\mathbb{L} = \Box e$. Let $[\![e]\!]^{\mathrm{TOP}} = \Theta \vdash E$. We have $[\![e]\!]^{\mathrm{TOP}}_{\zeta} = \emptyset \vdash \Box[\![e]\!]$ and $\Psi \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta}[[\![v]\!]^{\mathrm{TOP}}] = \Psi[\Theta_v \vdash V[\![e]\!]]$, which by lemma 1 reduces to $\Psi[\Theta_v + \Theta \vdash VE] = \Psi[[\![\mathbb{L}[v]]\!]^{\mathrm{TOP}}]$.
- $\mathbb{L} = \{s_v, X = \Box, s\}$. Let $[\![s_v]\!]^{\mathrm{TOP}} = \Theta'_v \vdash {S_v}'$, $[\![s]\!] = S$, and $[\![s]\!]^{\mathrm{TOP}} = \Theta' \vdash S'$.

  We have $\Psi \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta}[[\![v]\!]^{\mathrm{TOP}}] = \Psi[\Theta_v + \Theta'_v \vdash \{{S_v}', X = V, S\}]$, which by lemma 1 reduces to $\Psi[C] = \Psi[\Theta_v + \Theta'_v + \Theta' \vdash \{{S_v}', X = V, S'\}]$. If s is not evaluated, then C is exactly $[\![\mathbb{L}[v]]\!]^{\mathrm{TOP}}$. Otherwise, $\Psi[C]$ reduces by rule CONTEXT (ALLOCATE) to $\Psi[\Theta_v + \Theta'_v + \Theta' + \{l \mapsto \{{S_v}', X = V, S'\}\} \vdash l]$, which is exactly $\Psi[[\![\mathbb{L}[v]]\!]^{\mathrm{TOP}}]$.

□

Proposition 20 (Semi-compositionality for multiple lift contexts) *For all evaluation context* Ψ*,*

$$\Psi[[\![\mathbb{F}]\!]^{\mathrm{TOP}}_{\mathbf{Size}(v)}[[\![v]\!]^{\mathrm{TOP}}]] \longrightarrow^* \Psi[[\![\mathbb{F}[v]]\!]^{\mathrm{TOP}}].$$

Proof By induction on $\mathbb{F}$. If $\mathbb{F} = \Box$, there is nothing to prove. Otherwise, $\mathbb{F} = \mathbb{L}[\mathbb{F}']$. Let $\zeta = \mathbf{Size}(v)$ and $\zeta' = \mathbf{Size}(\mathbb{F}'[\Box_\zeta]) = \mathbf{Size}(\mathbb{F}'[v])$ (by hypothesis 1).

By proposition 19, as neither $\mathbb{F}'[\Box_\zeta]$ nor $\mathbb{F}'[v]$ are values, we have $[\![\mathbb{F}]\!]^{\mathrm{TOP}}_{\zeta} = [\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}']\!]^{\mathrm{TOP}}_{\zeta}]$ and $[\![\mathbb{F}[v]]\!]^{\mathrm{TOP}} = [\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}'[v]]\!]^{\mathrm{TOP}}]$.

By induction hypothesis,

$$\begin{aligned}&\Psi \circ [\![\mathbb{F}]\!]^{\mathrm{TOP}}_{\zeta}[[\![v]\!]^{\mathrm{TOP}}] \\ &= \Psi \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta'} \circ [\![\mathbb{F}']\!]^{\mathrm{TOP}}_{\zeta}[[\![v]\!]^{\mathrm{TOP}}] \\ &= \Psi \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}']\!]^{\mathrm{TOP}}_{\zeta}[[\![v]\!]^{\mathrm{TOP}}]] \\ &\longrightarrow^* \Psi \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}'[v]]\!]^{\mathrm{TOP}}] \\ &= \Psi[[\![\mathbb{L}]\!]^{\mathrm{TOP}}_{\zeta'}[[\![\mathbb{F}'[v]]\!]^{\mathrm{TOP}}]] \\ &= \Psi[[\![\mathbb{F}[v]]\!]^{\mathrm{TOP}}]\end{aligned}$$

□

Proposition 21 (Semi-compositionality for evaluation contexts) *For all evaluation context* Ψ*,*

$$\Psi[[\![\mathbb{E}]\!]^{\mathrm{TOP}}_{\mathbf{Size}(v)}[[\![v]\!]^{\mathrm{TOP}}]] \longrightarrow^* \Psi[[\![\mathbb{E}[v]]\!]^{\mathrm{TOP}}].$$

Proof By case analysis on $\mathbb{E}$.

- $\mathbb{E} = (b_v \vdash \mathbb{F})$. Let $\zeta = \mathbf{Size}(v)$ and $\zeta' = \mathbf{Size}(\mathbb{F}[\Box_\zeta]) = \mathbf{Size}(\mathbb{F}[v])$ (by hypothesis 1). We have

$$\begin{aligned}
&(\Psi \circ [\![\mathbb{E}]\!]_\zeta^{\mathrm{TOP}})[[\![v]\!]^{\mathrm{TOP}}] \\
&= \Psi \circ \mathbf{TOP}(b_v) \circ [\![\mathbb{F}]\!]_\zeta^{\mathrm{TOP}}[[\![v]\!]^{\mathrm{TOP}}] \\
&\longrightarrow^* \Psi \circ \mathbf{TOP}(b_v)[[\![\mathbb{F}[v]]\!]^{\mathrm{TOP}}] \\
&\qquad\qquad \text{(by proposition 20)} \\
&= \Psi[[\![b_v \vdash \mathbb{F}[v]]\!]^{\mathrm{TOP}}] \\
&= \Psi[[\![\mathbb{E}[v]]\!]^{\mathrm{TOP}}].
\end{aligned}$$

- $\mathbb{E} = (\mathbb{B}[\mathbb{F}] \vdash e)$, with $\mathbb{B} = (b_v, x = \Box, b)$. Let $\zeta = \mathbf{Size}(v)$ and $\zeta' = \mathbf{Size}(\mathbb{F}[\Box_\zeta]) = \mathbf{Size}(\mathbb{F}[v])$ (by hypothesis 1). Let also $b_0 = (x = \Box_{\zeta'}, b)$. We have

$$\begin{aligned}
&\Psi \circ [\![\mathbb{E}]\!]_\zeta^{\mathrm{TOP}}[[\![v]\!]^{\mathrm{TOP}}] \\
&= \Psi \circ \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ (\mathbf{TUp}(b_0)[\emptyset \vdash [\![e]\!]]) \circ [\![\mathbb{F}]\!]_\zeta^{\mathrm{TOP}}[[\![v]\!]^{\mathrm{TOP}}] \\
&\longrightarrow^* \Psi \circ \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ (\mathbf{TUp}(b_0)[\emptyset \vdash [\![e]\!]])[[\![\mathbb{F}[v]]\!]^{\mathrm{TOP}}] \\
&\qquad\qquad \text{(by proposition 20)}
\end{aligned}$$

  If $\mathbb{F}[v]$ is not a value, the obtained expression is exactly $\Psi[[\![\mathbb{E}[v]]\!]^{\mathrm{TOP}}]$. Otherwise, the obtained expression is a partial translation of $\mathbb{E}[v]$, so by proposition 15, it reduces to $\Psi[[\![\mathbb{E}[v]]\!]^{\mathrm{TOP}}]$, as expected.

□

6.2 Translation of access

In $\lambda_\circ$, the topmost binding is used as a heap, to store the values of variables. These values may then be copied when the corresponding bound variable is used in a strict context. In λ_{alloc}, heaps can only contain blocks, i.e. records and functions. Variables (or constants if the calculus featured them) cannot be stored in them. Instead, we have seen that they are substituted on the fly during the translation. This distinction makes the translation of access a bit weird.

Proposition 22 *If* $\mathbf{TOP}(b_v) = \Theta_a \vdash (\sigma, \eta)$, $b_v(x) = v$, *and* $[\![v]\!]^{\mathrm{TOP}} = \Theta_v \vdash V$, *then* $\Theta_v \subset \Theta_a$ *and* $(\sigma \circ \eta)(x) = V\{\sigma \circ \eta\}$.

Proof By induction on b_v.

- $b_v = \epsilon$. Contradicts $b_v(x) = v$.
- $b_v = (x = v, b_v{}')$ and $\mathbf{Size}(v) = n$. We have

$$\begin{array}{lcl}
[\![v]\!]^{\mathrm{TOP}} & = & \Theta_v \vdash l \\
\mathbf{TOP}(b_v{}') & = & \Theta'_a \vdash \sigma'\eta' \\
\mathbf{TOP}(b_v) & = & Th_v + \Theta'_a \vdash (\sigma', (\eta' + \{x \mapsto l\})) = \Theta_a \vdash (\sigma, \eta)
\end{array}$$

  Obviously, we have $\Theta_v \subset \Theta_a$. Furthermore, by syntactic correctness of b_v, $x \notin \mathbf{dom}(\sigma)$, so $(\sigma \circ \eta)(x) = \eta(x) = l = V = V\{\sigma \circ \eta\}$.

- $b_v = (x = v, b_v{}')$, with $\mathbf{Size}(v) = [?]$. We have

$$\begin{array}{lcl}
[\![v]\!]^{\mathrm{TOP}} & = & \emptyset \vdash y = \Theta_v \vdash V \\
\mathbf{TOP}(b_v{}') & = & \Theta'_a \vdash (\sigma', \eta') \\
\mathbf{TOP}(b_v) & = & \Theta'_a \vdash (\sigma' \circ \{x \mapsto y\}, \eta'),
\end{array}$$

  and therefore $(\sigma \circ \eta)(x) = y\{\eta'\} = V\{\eta\}$.

- $b_v = (y = v', b_v{}')$ and $\mathbf{Size}(v') = n$. We have

$$\begin{array}{lcl}
[\![v']\!]^{\mathrm{TOP}} & = & \Theta'_v \vdash l \\
\mathbf{TOP}(b_v{}') & = & \Theta'_a \vdash (\sigma', \eta') \\
\mathbf{TOP}(b_v) & = & \Theta'_a + \Theta'_v \vdash (\sigma', \eta' + \{y \mapsto l\}) = \Theta_a \vdash (\sigma, \eta).
\end{array}$$

  By induction hypothesis, $\Theta_v \subset \Theta'_a$, so $\Theta_v \subset \Theta'_a$. By induction hypothesis, $(\sigma' \circ \eta')(x) = V\{\eta'\}$, so $(\sigma \circ \eta)(x) = (\sigma' \circ \eta')(x)\{y \mapsto l\} = V\{\sigma' \circ \eta' \circ \{y \mapsto l\}\} = V\{\sigma \circ \eta\}$.

- $b_v = (y = v', b_v{}')$ and $\mathbf{Size}(v') = \mathit{undefined}$. We have

$$\begin{array}{lcl} [\![v']\!]^{\mathrm{TOP}} & = & \emptyset \vdash z \\ \mathbf{TOP}(b_v{}') & = & \Theta'_a \vdash (\sigma', \eta') \\ \mathbf{TOP}(b_v) & = & \Theta'_a \vdash (\sigma' \circ \{y \mapsto z\}, \eta') = \Theta_a \vdash (\sigma, \eta). \end{array}$$

  By induction hypothesis, $\Theta_v \subset \Theta'_a$, so $\Theta_v \subset \Theta'_a$. By induction hypothesis, $(\sigma' \circ \eta')(x) = V\{\eta'\}$. But by syntactic correctness of b_v, we know that y is not free in $b_v{}'$, so $y \notin \mathbf{cod}(\eta')$, and as we additionally have $y \notin \mathbf{dom}(\eta')$, we can deduce that $\{y \mapsto z\} \circ \eta' = \eta' \circ \{y \mapsto z\{\eta'\}\}$. So, we have

$$\begin{array}{l} (\sigma \circ \eta)(x) \\ = x\{\sigma' \circ \{y \mapsto z\} \circ \eta'\} \\ = x\{\sigma' \circ \eta' \circ \{y \mapsto z\{\eta'\}\}\} \\ = ((\sigma' \circ \eta')(x))\{y \mapsto z\{\eta'\}\} \\ = V\{\sigma' \circ \eta'\}\{y \mapsto z\{\eta'\}\} \\ = V\{\sigma' \circ \eta' \circ \{y \mapsto z\{\eta'\}\}\} \\ = V\{\sigma' \circ \{y \mapsto z\} \circ \eta'\} \\ = V\{\sigma \circ \eta\}. \end{array}$$

□

Proposition 23 (Access) *Let* $\Psi = [\![\mathbb{E}]\!]^{\mathrm{TOP}}_{\zeta} = \Theta \vdash \Phi[\sigma]$. *If* $\mathbb{E}(x) = v$ *and* $[\![v]\!]^{\mathrm{TOP}} = \Theta_v \vdash V$, *then* $\sigma(x) = V\{\sigma\}$ *and* $\Theta_v \subset \Theta$.

Proof By case analysis on the proof of $\mathbb{E}(x) = v$.

EA. $\mathbb{E} = b_v \vdash \mathbb{F}$, and $b_v(x) = v$. We have

$$[\![\mathbb{E}]\!]^{\mathrm{TOP}}_{\zeta} = \mathbf{TOP}(b_v) \circ [\![\mathbb{F}]\!]^{\mathrm{TOP}}_{\zeta}.$$

Let $\mathbf{TOP}(b_v) = \Theta_a \vdash (\sigma_a, \eta_a)$ and $[\![\mathbb{F}]\!]^{\mathrm{TOP}}_{\zeta} = \Theta' \vdash \Phi'[id]$. We can deduce $\sigma = \sigma_a \circ \eta_a$. By proposition 22, we have $\Theta_v \subset \Theta_a \subset \Theta$ and $(\sigma_a \circ \eta_a)(x) = V\{\sigma_a \circ \eta_a\}$, or in other words $\sigma(x) = V\{\sigma\}$, which is the expected result.

IA. $\mathbb{E} = (b_v, x = \mathbb{F}, b \vdash e)$. Then, $[\![\mathbb{E}]\!]^{\mathrm{TOP}}_{\zeta} = \mathbf{TDum}(x = \mathbb{F}[\Box_\zeta], b) \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(x = \mathbb{F}[\Box_\zeta], b)[\emptyset \vdash [\![e]\!]]$.

Let $\begin{array}{lcl} \mathbf{TDum}(x = \mathbb{F}[\Box_\zeta], b) & = & \Theta_d \vdash \eta_d \\ \mathbf{TOP}(b_v) & = & \Theta_a \vdash (\sigma_a, \eta_a) \\ \mathbf{TUp}(x = \mathbb{F}[\Box_\zeta], b)[\emptyset \vdash [\![e]\!]] & = & \Theta' \vdash \Phi'. \end{array}$

We have $\sigma = \sigma_a \circ \eta_a \circ \eta_d$. By proposition 22, $\Theta_v \subset \Theta_a$, so $\Theta_v \subset \Theta$. Furthermore, $(\sigma_a \circ \eta_a)(x) = V\{\sigma_a \circ \eta_a\}$, so $\sigma(x) = x\{\sigma_a \circ \eta_a \circ \eta_d\} = x\{\sigma_a \circ \eta_a\}\{\eta_d\} = V\{\sigma_a \circ \eta_a\}\{\eta_d\} = V\{\sigma\}$, as expected.

□

6.3 Translation of internal merging

Proposition 24 (Internal merging) *If* $b \vdash e \xrightarrow{\mathrm{IM}} b' \vdash e'$, *then* $[\![b \vdash e]\!]^{\mathrm{TOP}} \longrightarrow^* [\![b' \vdash e']\!]^{\mathrm{TOP}}$.

Proof Let $b \vdash e = (b_v, x = (\mathbf{let\ rec}\ b_1\ \mathbf{in}\ e_1), b_1 \vdash f)$, and $b' \vdash e' = (b_v, b_1, x = e_1, b_2 \vdash f)$. Let $b_0 = (x = (\mathbf{let\ rec}\ b_1\ \mathbf{in}\ e_1), b_2)$ and $b'_0 = (x = e_1, b_2)$.

We have $[\![b \vdash e]\!]^{\mathrm{TOP}} = \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(b_0)[\emptyset \vdash [\![f]\!]]$.

Let now $(x', \Phi') = \begin{cases} (x, \Box) \text{ if } \mathbf{Size}(e_1) = \mathbf{Size}(\mathbf{let\ rec}\ b_1\ \mathbf{in}\ e_1) = [?] \text{ (cf hypothesis 1)} \\ (y, \mathbf{update}\, x\, \Box) \text{ with } y \text{ fresh otherwise.} \end{cases}$

Let also $\Theta_1 \vdash E_1$ be defined as follows. If b_1 is evaluated, let $\Theta_1 \vdash E_1 = [\![e_1]\!]^{\mathrm{TOP}}$, and otherwise $\Theta_1 \vdash E_1 = \emptyset \vdash [\![e_1]\!]$. This way, we always have $[\![\mathbf{let\ rec}\ b_1\ \mathbf{in}\ e_1]\!]^{\mathrm{TOP}} = [\![b_1]\!]^{\mathrm{TOP}}[\Theta_1 \vdash E_1]$.

Finally, let $\Phi_1 = \emptyset \vdash \mathbf{let}\ x' = \Phi', \mathbf{Update}(b_2)\ \mathbf{in}\ [\![f]\!]$, and $b_1 = b_{v1}, b'_1$, where b'_1 does not begin with a value. We have

$$\begin{array}{l} \mathbf{TUp}(b_0)[\emptyset \vdash f] \\ = \Phi_1[[\![b_1]\!]^{\mathrm{TOP}}[\Theta_1 \vdash E_1]] \\ = \Phi_1 \circ \mathbf{TDum}(b'_1) \circ \mathbf{TOP}(b_{v1}) \circ \mathbf{TUp}(b'_1)[\Theta_1 \vdash E_1]. \end{array}$$

But the context $\mathbf{TDum}(b'_1) \circ \mathbf{TOP}(b_{v1})$ is a weak evaluation context, and the domain of its substitution only concerns variables in the domain of b_1, which are disjoint from free variables in b_2, f, x by the side condition to the rule IM. Therefore, this context commutes with Φ_1, and

$$\begin{aligned}&\mathbf{TUp}(b_0)[\emptyset \vdash f]\\ &= \mathbf{TDum}(b'_1) \circ \mathbf{TOP}(b_{v1}) \circ \Phi_1 \circ \mathbf{TUp}(b'_1)[\Theta_1 \vdash E_1].\end{aligned}$$

Now, if b_1 is not fully evaluated, the two translation are semantically identical. But if b_1 is fully evaluated, i.e. $b'_1 = \epsilon$, then $[\![b' \vdash e']\!]^{\mathrm{TOP}}$ translates with the TOP translation until e_1, and possibly further, if e_1 is a value too. We distinguish the two cases.

1. b_1 is not fully evaluated. Let $\mathbf{TUp}(b'_1) = \Theta'_1 \vdash B'_1$. We have $\Theta_1 \vdash E_1 = \emptyset \vdash [\![e_1]\!]$ and with $\phi = \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ \mathbf{TDum}(b'_1) \circ \mathbf{TOP}(b_{v1})$,

$$\begin{aligned}&[\![b \vdash e]\!]^{\mathrm{TOP}}\\ &= \phi[\Theta'_1 \vdash \mathbf{let}\ x' = \mathbf{let}\ B'_1\ \mathbf{in}\ [\![e_1]\!][,]\,\mathbf{Update}(b_2)\ \mathbf{in}\ [\![f]\!]]\\ &\xrightarrow{\text{LIFT}} \phi[\Theta'_1 \vdash \mathbf{let}\ B'_1\ \mathbf{in}\ \mathbf{let}\ x' = [\![e_1]\!][,]\,\mathbf{Update}(b_2)\ \mathbf{in}\ [\![f]\!]]\\ &\xrightarrow{\text{EM}} \phi[\Theta'_1 \vdash \mathbf{let}\ B'_1, x' = [\![e_1]\!][,]\,\mathbf{Update}(b_2)\ \mathbf{in}\ [\![f]\!]]\\ &= \phi[\Theta'_1 \vdash \mathbf{let}\ B'_1, x' = [\![e_1]\!][,]\,\mathbf{Update}(b_2)\ \mathbf{in}\ [\![f]\!]]\\ &= \phi \circ \mathbf{TUp}(b'_1, b'_0)[\emptyset \vdash [\![f]\!]].\end{aligned}$$

   But let us now examine ϕ a bit $\mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ \mathbf{TDum}(b'_1) \circ \mathbf{TOP}(b_{v1})$. First, notice that $\mathbf{TDum}(b_0) = \mathbf{TDum}(b'_0)$, by hypothesis 1.

   Then, $\mathbf{TOP}(b_v)$ and $\mathbf{TDum}(b'_1)$ are two weak evaluation contexts, and the domain of the substitution of $\mathbf{TDum}(b'_1)$ is included in $\mathbf{dom}(b'_1)$, which is disjoint from the free variables of b_v, so if $\mathbf{TDum}(b'_1) = \Theta'_d \vdash \eta'_d$, then $\mathbf{TOP}(b_v) \circ \mathbf{TDum}(b'_1) = \eta'_d \circ \mathbf{TOP}(b_v) \circ \mathbf{TDum}(b'_1)$. Moreover, η'_d is a variable allocation, and is therefore idempotent, so we can apply proposition 12 to obtain

$$\begin{aligned}\phi &= \mathbf{TDum}(b'_0) \circ \mathbf{TDum}(b'_1) \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1})\\ &= \mathbf{TDum}(b'_0, b'_1) \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1})\\ &= \mathbf{TDum}(b'_1, b'_0) \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1}).\end{aligned}$$

   Furthermore, $\mathbf{TOP}(b_{v1}) = \Theta_{b_{v1}} \vdash (\sigma_{b_{v1}}, \eta_{b_{v1}})$. As $\eta_{b_{v1}}$ is idempotent, we have $\mathbf{TOP}(b_{v1}) = \eta_{b_{v1}} \circ \mathbf{TOP}(b_{v1})$. But we know that the domain of $\eta_{b_{v1}}$ is disjoint from the free variables of $\mathbf{TOP}(b_v)$, so $\mathbf{TOP}(b_v) \circ \eta_{b_{v1}} = \eta_{b_{v1}} \circ \mathbf{TOP}(b_v)$, and therefore $\phi = \mathbf{TDum}(b'_1, b'_0) \circ \eta_{b_{v1}} \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1})$. But by corollary 4, $\eta_{b_{v1}} \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1}) = \mathbf{TOP}(b_v, b_{v1})$, so $\phi = \mathbf{TDum}(b'_1, b'_0) \circ \mathbf{TOP}(b_v, b_{v1})$.

   Finally, we obtain that

$$\begin{aligned}[\![b \vdash e]\!]^{\mathrm{TOP}} &= \mathbf{TDum}(b'_1, b'_0) \circ \mathbf{TOP}(b_v, b_{v1}) \circ \mathbf{TUp}(b'_1, b'_0)[\emptyset \vdash [\![f]\!]]\\ &= [\![b_v, b_{v1}, b'_1, b'_0]\!]^{\mathrm{TOP}}[\emptyset \vdash [\![f]\!]]\\ &= [\![b_v, b_1, x = e_1, b_2]\!]^{\mathrm{TOP}}[\emptyset \vdash [\![f]\!]]\\ &= [\![b' \vdash e']\!]^{\mathrm{TOP}}.\end{aligned}$$

2. b_1 is fully evaluated. We have $[\![b \vdash e]\!]^{\mathrm{TOP}} = \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1}) \circ \Phi_1[\Theta_1 \vdash E_1]$.

   Let $\mathbf{TOP}(b_{v1}) = \Theta_{b_{v1}} \vdash (\sigma_{b_{v1}}, \eta_{b_{v1}})$. We know that $\eta_{b_{v1}}$ is idempotent, so $\mathbf{TOP}(b_{v1}) = \eta_{b_{v1}} \circ \mathbf{TOP}(b_{v1})$. As above, $\mathbf{dom}(\eta_{b_{v1}}) \perp \mathbf{FV}(\mathbf{TOP}(b_v))$, so $\mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1}) = \eta_{b_{v1}} \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_{v1})$, in which by corollary 4 we recognize $\mathbf{TOP}(b_v, b_{v1})$.

   Therefore, $[\![b \vdash e]\!]^{\mathrm{TOP}} = \mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v, b_{v1}) \circ \Phi_1[\Theta_1 \vdash E_1]$.

   But we notice that $\Phi_1[\Theta_1 \vdash E_1] = \mathbf{TUp}(b'_0)[\emptyset \vdash [\![f]\!]]$. And by hypothesis 1, $\mathbf{TDum}(b_0) = \mathbf{TDum}(b'_0)$. Let $\mathbf{TDum}(b_0) = \Theta_{b_0} \vdash \eta_{b_0}$. By proposition 12, we have $\mathbf{TDum}(b_0) \circ \mathbf{TOP}(b_v, b_{v1}) = \eta_{b_0} \circ \mathbf{TOP}(b_v, b_{v1}) \circ \mathbf{TDum}(b'_0)$, so $[\![b \vdash e]\!]^{\mathrm{TOP}} = \eta_{b_0} \circ \mathbf{TOP}(b_v, b_{v1}) \circ \mathbf{TDum}(b'_0) \circ \mathbf{TUp}(b'_0)[\emptyset \vdash [\![f]\!]]$.

   Let $b'_0 = (b_{v0}, b''_0)$, with b''_0 not beginning with a value. By proposition 13, $[\![b \vdash e]\!]^{\mathrm{TOP}} \longrightarrow^* \eta_{b_0} \circ \mathbf{TOP}(b_v, b_{v1}) \circ \mathbf{TDum}(b''_0) \circ \mathbf{TOP}(b_{v0}) \circ \mathbf{Update}(b''_0)[\emptyset \vdash [\![f]\!]]$.

   But if $\mathbf{TDum}(b_{v0}) = \Theta_{b_{v0}} \vdash \eta_{b_{v0}}$ and $\mathbf{TDum}(b''_0) = \Theta_{b''_0} \vdash \eta_{b''_0}$, then $\eta_{b_0} = \eta_{b_{v0}} + \eta_{b''_0}$, so by proposition 12, the obtained expression is equal to $\eta_{b_{v0}} \circ \mathbf{TDum}(b''_0) \circ \mathbf{TOP}(b_v, b_{v1}) \circ \mathbf{TOP}(b_{v0}) \circ \mathbf{Update}(b''_0)[\emptyset \vdash [\![f]\!]]$. But $\eta_{b_{v0}}$ commutes with $\mathbf{TDum}(b''_0)$, so we obtain $\mathbf{TDum}(b''_0) \circ \eta_{b_{v0}} \circ \mathbf{TOP}(b_v, b_{v1}) \circ \mathbf{TOP}(b_{v0}) \circ \mathbf{Update}(b''_0)[\emptyset \vdash [\![f]\!]]$, which by corollary 4 is equal to $\mathbf{TDum}(b''_0) \circ \eta_{b_{v0}} \circ \mathbf{TOP}(b_v, b_{v1}, b_{v0}) \circ \mathbf{Update}(b''_0)[\emptyset \vdash [\![f]\!]]$, which is exactly $[\![b' \vdash e']\!]^{\mathrm{TOP}}$.

□

• **Evaluated binding contexts**
$\mathbb{B}_v ::= b_{v1}, x = \square, b_{v2}$ with $\mathbf{Depth}(b_{v1}, x = \square, b_{v2})$ defined as $1+ \mid b_{v1} \mid$
• **Depth of an evaluation context**

$$\begin{array}{rcl} \mathbf{Depth}(\square) & = & 0 \\ \mathbf{Depth}(\mathbb{L}\,[\mathbb{F}]) & = & 1 + \mathbf{Depth}(\mathbb{F}) \\ \mathbf{Depth}(b_v \vdash \mathbb{F}) & = & 1+ \mid b_v \mid + \mathbf{Depth}(\mathbb{F}) \\ \mathbf{Depth}(\mathbb{B}_v\,[\mathbb{F}] \vdash e) & = & \mathbf{Depth}(\mathbb{B}_v) + \mathbf{Depth}(\mathbb{F}) \end{array}$$

• **Measuring the number of let rec nodes**
$\mu_{\mathbf{l}}(e)$ is the number of **let rec** nodes not under a λ in e (same for configurations).
• **Measuring the depth of the let rec to lift** (same for configurations)

$$\begin{array}{rcl} \mu_{\mathbf{d}}(\mathbb{F}[\mathbb{L}[\textbf{let rec } b_v \textbf{ in } e]]) & = & 1 + \mathbf{Depth}(\mathbb{F}) \\ \mu_{\mathbf{d}}(e) & = & 0 \text{ otherwise} \end{array}$$

well defined since the sum of the depths of **let rec** nodes strictly decreases.
• **Measuring the binding level of the hot variable**
$\mu_{\mathbf{b}}(e)$ is the depth of the binder of the hot variable, if any:

$$\begin{array}{rcll} \mu_{\mathbf{b}}(\mathbb{B}_v\,[v], y = \mathbb{F}\,[x], b \vdash e) & = & \mathbf{Depth}(\mathbb{B}_v) & \text{if } (\mathbb{B}_v\,[v])(x) = v \\ \mu_{\mathbf{b}}(\mathbb{B}_v\,[v] \vdash \mathbb{F}\,[x]) & = & \mathbf{Depth}(\mathbb{B}_v) & \text{if } (\mathbb{B}_v\,[v])(x) = v \\ \mu_{\mathbf{b}}(e) & = & 0 & \text{otherwise} \end{array}$$

• **Measure** $\mu_{\mathbf{e}}(e) = (\mu_{\mathbf{l}}(e), \mu_{\mathbf{d}}(e))$ (lexicographically ordered).
$\mu(c) = (\mu_{\mathbf{l}}(c), \mu_{\mathbf{d}}(c), \mu_{\mathbf{b}}(c))$

Figure 21: Measure

6.4 Simulation

Due to their different ways of handling bindings, the two calculus $\lambda_\circ$ 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 any expressions of the shape $\mathbb{L}\,[\textbf{let rec } b_v \textbf{ in } e]$ and $\textbf{let rec } b_v \textbf{ in } \mathbb{L}\,[e]$. 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. The only problem with this is that in some cases an infinite reduction sequence in $\lambda_\circ$ could be translated as an empty one in λ_{alloc}, thus possibly changing the infinite looping observable behaviour. In order to ensure that this doesn't happen, we prove that such silent reduction steps cannot happen indefinitely. For this, we introduce a measure on expressions and configurations that strictly decreases during silent reductions steps. Its definition is given in figure 21.

It first defines two functions from expressions to $\mathbb{N}$. The first, $\mu_{\mathbf{l}}$, is the number of **let rec** nodes not under a lambda in the given expression. The second, $\mu_{\mathbf{d}}$ is the depth of the **let rec** node to lift in the given expression, if any. Formally, if e is of the shape $\mathbb{F}\,[\mathbb{L}\,[\textbf{let rec } b \textbf{ in } f]]$, then the **let rec** node can be lifted by rule LIFT, so the result is the depth of the context $\mathbb{F}\,[\mathbb{L}\,]$, or 1 plus the depth of $\mathbb{F}$.

The functions $\mu_{\mathbf{l}}$ and $\mu_{\mathbf{d}}$ form a measure $\mu_{\mathbf{e}}$ on expressions, defined by $\mu_{\mathbf{e}} = (\mu_{\mathbf{l}}, \mu_{\mathbf{d}})$, ordered lexicographically.

Moreover, these two functions are straightforwardly extended to configurations, replacing $\mathbb{F}$ with $\mathbb{E}$ for the second definition.

A third function $\mu_{\mathbf{b}}$ is defined, but only on configurations, giving the depth of the binder for the *hot* variable, if any. We say that x is the hot variable in c if c is of the shape $\mathbb{E}\,[\mathbb{N}\,[x]]$. Then $\mu_{\mathbf{b}}(e)$ is the depth at which x is bound in $\mathbb{E}$. Formally, we define evaluated binding contexts as binding contexts of the shape $b_{v1}, x = \square, b_{v2}$, and their depth as 1 plus the cardinal of b_{v1}. Then the depth of multiple lift contexts is defined as the number of nested lift contexts, and the depth of evaluation contexts is defined accordingly.

A property of this measure is that it is monotone through contextual closure.

Proposition 25 *If $\mu_{\mathbf{e}}(e) > \mu_{\mathbf{e}}(e')$, then for any evaluation context $\mathbb{E}$, $\mu(\mathbb{E}\,[e]) > \mu(\mathbb{E}\,[e'])$.*

Proof The property clearly holds for both measures $\mu_{\mathbf{l}}$ and $\mu_{\mathbf{d}}$, thus for their lexicographic product as well. □

Lemma 3 (Contraction simulated) *If $e \rightsquigarrow_c e'$, then $[\![e]\!]^{\mathrm{TOP}} \longrightarrow^+ [\![e']\!]^{\mathrm{TOP}}$ or $[\![e]\!]^{\mathrm{TOP}} = [\![e']\!]^{\mathrm{TOP}}$ and for any $\mathbb{E}$, $\mu(\mathbb{E}[e]) > \mu(\mathbb{E}[e'])$.*

Proof By case analysis on the applied rule.

Beta. $e = ((\lambda x.f)v)$, and $e' = letrecinx = vf$. Let $[\![v]\!]^{\mathrm{TOP}} = \Theta_v \vdash V$. We have $[\![e]\!]^{\mathrm{TOP}} = \Theta_v + \{l \mapsto (\lambda x.[\![f]\!])\} \vdash lV$, which reduces by rule Beta to $\Theta_v + \{l \mapsto (\lambda x.[\![f]\!])\} \vdash f\{x \mapsto V\}$.

Let us now calculate $\mathbf{TOP}(x = v)$.

- If $\mathbf{Size}(v) = [?]$, then $\Theta_v \vdash V = \emptyset \vdash V$, and $\mathbf{TOP}(x = v) = \emptyset \vdash (x \mapsto V, id)$; so $[\![x = v]\!]^{\mathrm{TOP}} = \emptyset \vdash \Box[x \mapsto V] = \Theta_v \vdash \Box[x \mapsto V]$.
- Otherwise, $\Theta_v \vdash V = \Theta_v \vdash l$, and $\mathbf{TOP}(x = v) = \Theta_v \vdash (id, x \mapsto l)$; so $[\![x = v]\!]^{\mathrm{TOP}} = \Theta_v \vdash \Box[x \mapsto l] = \Theta_v \vdash \Box[x \mapsto V]$.

So, in both cases, we have $[\![x = v]\!]^{\mathrm{TOP}} = \Theta_v \vdash \Box[x \mapsto V]$. Therefore, $[\![x = v]\!]^{\mathrm{TOP}}$ reduces to $[\![x = v]\!]^{\mathrm{TOP}}[[\![f]\!]]$, which by lemma 1 reduces to $[\![x = v]\!]^{\mathrm{TOP}}[[\![f]\!]^{\mathrm{TOP}}]$, which is exactly $[\![e']\!]^{\mathrm{TOP}}$.

Project. $e = \{s_v\}.X$ and $e' = s_v(X)$. Let $s_v = (X_1 = v_1 \dots X_n = v_n)$, $X = X_{i_0}$, and for each i, $[\![v_i]\!]^{\mathrm{TOP}} = \Theta_i \vdash V_i$. We have $[\![s_v]\!]^{\mathrm{TOP}} = \biguplus_{1 \le i \le n} \Theta_i \vdash (X_1 = V_1 \dots X_n = V_n)$, and $[\![e]\!]^{\mathrm{TOP}} = \biguplus_{1 \le i \le n} \Theta_i + \{l \mapsto \{X_1 = V_1 \dots X_n = V_n\}\} \vdash l.X$. By rule Project, it reduces to $\biguplus_{1 \le i \le n} \Theta_i + \{l \mapsto \{X_1 = V_1 \dots X_n = V_n\}\} \vdash V_{i_0}$, which by rule GC reduces to $\Theta_{i_0} \vdash V_{i_0}$, which is exactly $[\![e']\!]^{\mathrm{TOP}}$.

Lift. $e = \mathbb{L}[\mathbf{let\ rec}\ b\ \mathbf{in}\ f]$ and $e' = \mathbf{let\ rec}\ b\ \mathbf{in}\ \mathbb{L}[f]$.

- If b is evaluated, then $[\![e]\!]^{\mathrm{TOP}} = [\![\mathbb{L}]\!]^{\mathrm{TOP}} \circ \mathbf{TOP}(b)[[\![f]\!]^{\mathrm{TOP}}]$. Let $\mathbf{TOP}(b) = \Theta \vdash (\Box, \sigma)$. In the context $[\![\mathbb{L}]\!]^{\mathrm{TOP}} \circ \mathbf{TOP}(b)$, the only substitution is σ, whose domain is $\mathbf{dom}(b)$, which by the side condition to the Lift rule is disjoint from the free variables of $\mathbb{L}$, so the contexts commute, and $[\![e]\!]^{\mathrm{TOP}} = \mathbf{TOP}(b) \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}[[\![f]\!]^{\mathrm{TOP}}] = [\![e']\!]^{\mathrm{TOP}}$.
- If b is not evaluated, then $b = b_v, b'$, with b' non empty and not beginning with a value. We have $[\![e]\!]^{\mathrm{TOP}} = [\![\mathbb{L}]\!]^{\mathrm{TOP}} \circ \mathbf{TDum}(b') \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(b')[\emptyset \vdash [\![f]\!]]$. But as above, the context $[\![\mathbb{L}]\!]^{\mathrm{TOP}}$ has not substitution and is not affected by the ones of $\mathbf{TDum}(b')$, $\mathbf{TOP}(b_v)$, and $\mathbf{TUp}(b')$. So $[\![e]\!]^{\mathrm{TOP}} = \mathbf{TDum}(b') \circ \mathbf{TOP}(b_v) \circ \mathbf{TUp}(b') \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}[\emptyset \vdash [\![f]\!]] = [\![e']\!]^{\mathrm{TOP}}$.

This is the only case where the two translations are directly equal. We thus have to show that $\mu_{\mathbf{d}}(e) > \mu_{\mathbf{d}}(e')$. And indeed $\mu_{\mathbf{d}}(e) = \mu_{\mathbf{d}}(\mathbb{L}[\mathbf{let\ rec}\ b\ \mathbf{in}\ f]) = 2 + 0$, whereas $\mu_{\mathbf{d}}(e') = \mu_{\mathbf{d}}(\mathbf{let\ rec}\ b\ \mathbf{in}\ \mathbb{L}[f]) = 0$. Conclude by proposition 25.

□

There is a last difficulty lying in the way to the theorem of simulation, due to different sharing properties of the two calculi. 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

$$C = \left\{ \begin{array}{l} l_1 \mapsto \lambda y.y, \\ l_2 \mapsto \{X = l_1\} \end{array} \right\} \vdash (l_2.X)l_2.$$

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

$$C' = \left\{ \begin{array}{l} l_1 \mapsto \lambda y.y, \\ l_2 \mapsto \{X = l_1\}, \\ l_3 \mapsto \lambda y.y, \\ l_4 \mapsto \{X = l_3\} \end{array} \right\} \vdash (l_4.X)l_2.$$

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 (see below), such a reduction step is necessarily followed by a Beta or a Project step. In our example, a Project step occurs, that destroys the copied record: 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, it is immediately destroyed by a 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 reductions steps to be simulated by a single one.

But this is not yet sufficient. Indeed, 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'':

$$C'' = \left\{ \begin{array}{l} l_1 \mapsto \lambda y.y, \\ l_2 \mapsto \{X = l_1\}, \\ l_3 \mapsto \lambda y.y \end{array} \right\} \vdash l_3 l_2.$$

Our solution to this problem consists in only considering 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.

$$\frac{\exists i, e_i \notin \mathbf{Vars} \qquad \forall i, j, x_i \notin \mathbf{FV}(e_j)}{\{X_1 = e_1 \ldots X_n = e_n\} \xrightarrow{\mathbf{R}} \mathbf{let\ rec}\ x_1 = e_1 \ldots x_n = e_n\ \mathbf{in}\ \{X_1 = x_1 \ldots X_n = x_n\}} \quad (\text{NAMEFIELDS})$$

This process necessarily terminates since the number of records not containing only variables stricly decreases. The reduction rules of $\lambda_\circ$ obviously preserve the **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.

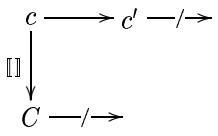
We can now state our final theorem. A $\lambda_\circ$ configuration is said *stuck on a free variable* when it is of the shape $\mathbb{E}[\mathbb{N}[x]]$ and $\mathbb{E}(x)$ is undefined. This definition is extended to λ_{alloc} configurations (replace $\mathbb{E}$ with Ψ). We say that a configuration is faulty if it is in normal form and is not a valid answer and is not stuck on a free variable. Roughly, the theorem states that if a configuration c reduces to another one c', then

- either c' is faulty and so is the translation of c,
- or the translation of c reduces to the one of c',
- or c' itself reduces to c'', such that the translation of c reduces to the one of c'',
- or c and c' are translated to the same configuration, but $\mu(c) > \mu(c')$.

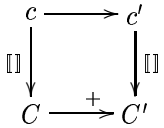
This complicated result is due to the fact that $\lambda_\circ$ first needs to duplicate a function before to apply it, and to duplicate a record before to select a component from it, and to the fact that the TOP translation identifies some configurations, by performing some lifting and merging steps by itself.

Theorem 1 (Small steps encoding) *For all **R**-normal configuration c, if $c \longrightarrow c'$ and $[\![c]\!]^{\mathrm{TOP}} = C$, then one of the four situations below holds:*

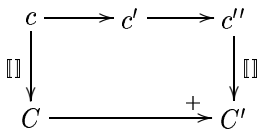
1. *Either c' is faulty, and then C is faulty too ;*



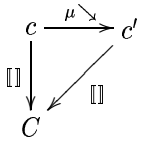
2. *or there exists C' such that $[\![e']\!] = C'$ and $C \longrightarrow^+ C'$;*



3. *or there exists c'', C' such that $[\![c'']\!] = C'$ and $C \longrightarrow^+ C'$;*



4. *or* $[\![c']\!] = C$ *directly, and* $\mu(c) > \mu(c')$



Proof By case analysis on the applied rule.

Context. By lemma 3.

IM. By proposition 24, noting that the number of **let rec** nodes decreases by one when applying the rule.

EM. $c = b_v \vdash \textbf{let rec } b \textbf{ in } e$ and $c' = b_v, b \vdash e$. Let us now define C_1 by $\emptyset \vdash [\![e]\!]$ if b is not evaluated, and $[\![e]\!]^{\text{TOP}}$ otherwise. Then $[\![c]\!]^{\text{TOP}} = \mathbf{TOP}(b_v) \circ [\![b]\!]^{\text{TOP}}[C_1]$. Let $b = b_v', b'$, where b' does not begin with a value. We have $[\![c]\!]^{\text{TOP}} = \mathbf{TOP}(b_v) \circ \mathbf{TDum}(b') \circ \mathbf{TOP}(b_v') \circ \mathbf{TUp}(b')[C_1]$. But the substitution of the context $\mathbf{TDum}(b')$ does not affect $\mathbf{TOP}(b_v)$ and conversely the substitution of $\mathbf{TOP}(b_v)$ does not affect $\mathbf{TDum}(b')$, so the two contexts commute. But then $\mathbf{TOP}(b_v)$ is next to $\mathbf{TOP}(b_v')$. Let η be the substitution of $\mathbf{TDum}(b_v')$. It does not affect $\mathbf{TOP}(b_v)$, by the side condition to the EM rule, so $\mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_v') = \eta \circ \mathbf{TOP}(b_v) \circ \mathbf{TOP}(b_v')$, which by corollary 4 is equal to $\mathbf{TOP}(b_v, b_v')$. Therefore, $[\![c]\!]^{\text{TOP}} = \mathbf{TDum}(b') \circ \mathbf{TOP}(b_v, b_v') \circ \mathbf{TUp}(b')[C_1] = [\![b_v, b]\!]^{\text{TOP}}[C_1]$. The number of **let rec** nodes again decreases by one.

Subst. $c = \mathbb{E}[\mathbb{N}[x]]$, $c' = \mathbb{E}[\mathbb{N}[v]]$, and $\mathbb{E}(x) = v$. Let $\Psi = [\![\mathbb{E}]\!]^{\text{TOP}} = \Theta \vdash \Phi[\sigma]$.

- If v is a variable y, then $[\![v]\!]^{\text{TOP}} = \emptyset \vdash y$, and by proposition 23, $\sigma(x) = y\{\sigma\}$, so $[\![c]\!]^{\text{TOP}} = [\![c']\!]^{\text{TOP}}$. But, the depth of the binder of the hot variable, from the depth of $x = y$ in $\mathbb{E}$, becomes either an upper $y = v'$ definition, or the depth 0, if y is not defined by $\mathbb{E}$, so $\mu(c) > \mu(c')$.
- If c' is faulty, i.e. either $\mathbb{N} = \Box v'$ and v is a record, or $\mathbb{N} = \Box.X$ and v is a function or a record with no X field, then C is faulty too.
- If $v = \lambda y.e$ and $\mathbb{N} = \Box v'$, then $c' \longrightarrow c'' = \mathbb{E}[\textbf{let rec } y = v' \textbf{ in } e]$.
  Let $[\![v']\!]^{\text{TOP}} = \Theta_v' \vdash V'$. Let $\phi = \Theta_v' \vdash \Box[id]$. We have $C = \Psi \circ \phi[lV']$.
  But by proposition 23, the location $l = \sigma(x)$ is such that $\Theta(l) = \lambda y.[\![e]\!]$. Therefore, C reduces by rule Context (Beta) to $\Psi \circ \phi[[\![e]\!]\{y \mapsto V'\}]$. By lemma 1, this reduces to $\Psi \circ \phi[[\![e]\!]^{\text{TOP}}\{y \mapsto V'\}]$.
  Let now $\phi' = \phi \circ \{y \mapsto V'\}$. The obtained configuration can be written $\Psi \circ \phi'[[\![e]\!]^{\text{TOP}}]$.
  But $\mathbf{TOP}(y = v') = \Theta_v' \vdash \Box[y \mapsto V'] = \phi'$, so $[\![\textbf{let rec } y = v' \textbf{ in } e]\!]^{\text{TOP}} = \phi'[[\![e]\!]^{\text{TOP}}]$, and the obtained term can also be written $[\![\mathbb{E}]\!]^{\text{TOP}}[[\![\textbf{let rec } y = v' \textbf{ in } e]\!]^{\text{TOP}}]$, which by proposition 21, reduces to $[\![\mathbb{E}[\textbf{let rec } y = v' \textbf{ in } e]]\!]^{\text{TOP}}$, which is exactly $[\![c'']\!]^{\text{TOP}}$.
- If $v = \{s_v\}$, $\mathbb{N} = \Box.X$, with $X \in \mathbf{dom}(s_v)$, then $c' \longrightarrow c'' = \mathbb{E}[s_v(X)]$.
  By hypothesis, c is in **R**-normal form, so there exist names $X_1 \ldots X_n$ and variables $x_1 \ldots x_n$ such that $s_v = (X_1 = x_1 \ldots X_n = x_n)$. Then, s_v can be viewed as a record of λ_{alloc}, and $[\![v]\!]^{\text{TOP}} = \{l \mapsto \{s_v\}\} \vdash l$.
  By proposition 23, we have $\sigma(x) = l$ and $\Theta(l) = \{s_v\}$. We have $[\![c]\!]^{\text{TOP}} = \Psi[x.X] = \Psi[l.X]$. As c reduces to c', there exists an index i_0 such that $X = X_{i_0}$. So, $[\![c]\!]^{\text{TOP}}$ reduces in one Project step to $\Psi[x_{i_0}]$, which is $[\![\mathbb{E}]\!]^{\text{TOP}}[[\![x_{i_0}]\!]^{\text{TOP}}]$, so by lemma 1, it reduces to $[\![\mathbb{E}[x_{i_0}]]\!]^{\text{TOP}}$, which is exactly the translation of c''.

□

Eventually, we state a less precise theorem, more like what we would obtain with big step semantics.

Theorem 2 (Big steps encoding)

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

Proof For items 1 and 2, notice that $\emptyset \vdash [\![e]\!]$ reduces to $[\![e]\!]^{\mathrm{TOP}}$, and then reason by induction on the length of the reduction sequence. For item 3, by contrapositive: we know that there is a reduction sequence in λ_{alloc} simulating the one in $\lambda_\circ$, but it could be of phantom steps, i.e. the same configuration could be a translation for all steps. However this would contradict the strict decreasing of the measure, which is of course bounded by 0. For item 4, the reduction leading to the configuration stuck on a free variable is simulated, and the reached configuration being the translation of a stuck configuration is also stuck. □

The initial goal here was to prove the correctness of our compilation scheme, but in fact we have a completeness result for free.

Theorem 3 (Big steps completeness)

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.

Proof There are four possible final states for a configuration: it can reduce to a value, or it can get stuck on a free variable, or it can go wrong, or it can loop. We know that if a configuration $\emptyset \vdash e$ reaches a final state, then so does $[\![\emptyset \vdash e]\!]^{\mathrm{TOP}}$. But the four possible final states are mutually exclusive. Therefore, if the translation of an expression reaches a final state, then the original configuration necessarily reaches the same one. □

Remark 1 (Free variables) *Free variables do not appear during reduction, and the cases where the evaluation gets stuck on a free variable do not occur if the initial expression is closed.*

7 Related work and conclusion

Cyclic explicit substitutions In [13], Rose defines a calculus with mutually recursive definitions, where the dedicated construct for recursion is presented as *explicit cyclic substitution*, referring to the explicit substitutions of Lévy et al. [1]. Instead of lifting recursive bindings to the top of terms as we do, the 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 really using each other. In this respect, it is less powerful than $\lambda_\circ$. Besides, it does not impose size constraints on definitions, but is also not concerned with data representation.

Lescanne et al. [3] study sharing and different evaluation strategies, with 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 [3] 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, as a quite straightforward specialization of it. Thus, our work can be seen as importing the internal substitution rule IA from equational theory to language design. Nevertheless, the concerns are different: we deal with implementation and data representation, while Ariola et al. rather examine confluence, sharing and different evaluation strategies, including strong reduction (reduction under λ-abstraction).

let rec for objects and mixin modules Boudol's construct [4], or Hirschowitz and Leroy's [8], are different 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 an extensive use of recursive definitions such as **let rec** $o = \mathit{generator}(o)$ **in** e. Such definitions are impossible in $\lambda_\circ$. However, $\lambda_\circ$ allows to define in the same binding some recursive values, followed by computations using these values. The semantics of mixin modules [9] requires complex sequences of alternate recursive and non-recursive bindings, which are trivial to write in $\lambda_\circ$. On the whole, the loss of flexibility for valid recursive definitions allows to improve efficiency, thanks to the loss of additional indirections.

We believe that it is possible to combine the ideas of [4] and [9]. Consider a language where a recursive definition can be of any shape, and can now be syntactically annotated with integers representing its expected size. This language can be compiled exactly as $\lambda_\circ$, but it features a more powerful **let rec** construct. The idea should be seen as a compilation technique for Boudol's objects and Hirschowitz and Leroy's mixin modules, where the necessary size informations are statically available.

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 1996*, 1996.

[4] G. Boudol. The recursive record semantics of objects revisited. In D. Sands, editor, *Europ. Symp. on Progr. 2001*, 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. 2002*, volume 2305 of *LNCS*, pages 6–20, 2002.

[9] T. Hirschowitz, X. Leroy, and J. B. Wells. A reduction semantics for call-by-value mixin modules. Technical report, INRIA, 2003.

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

[11] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http://caml.inria.fr/`, 1996–2003.

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

[13] K. H. Rose. Explicit cyclic substitution. Unpublished, Mar. 1993.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399