Compilation of extended recursion in call-by-value functional languages

Tom Hirschowitz, Xavier Leroy, J. B. Wells

Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Compilation of extended recursion in call-by-value functional languages

Tom Hirschowitz · **Xavier Leroy** · **J. B. Wells**

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

Keywords Compilation · Recursion · Semantics · Functional languages

1 Introduction

1.1 The need for extended recursion

Functional languages usually feature mutually recursive definition of values, for example via the `letrec` construct in Scheme, `let rec` in Caml, `val rec` and `fun` in Standard ML, or recursive equations in Haskell. Beyond syntax, functional languages differ also in the kind of expressions they support as right-hand sides of mutually recursive definitions. For instance, Haskell [25] allows arbitrary expressions as right-hand sides of recursive definitions, while Standard ML [22] only allows syntactic λ-abstractions, and OCaml [21] allows both λ-abstractions and limited forms of constructor applications.

The range of allowed right-hand sides crucially depends on the evaluation strategy of the language. Call-by-name or lazy languages such as Haskell naturally implement arbitrary recursive definitions: the on-demand unwinding of the recursive definition performed by lazy

Work partially supported by EPSRC grant GR/R 41545/01. This article is a revised and much extended version of [15].

Tom Hirschowitz
Department of Mathematics, University of Savoie, Campus Scientifique, 73376 Le Bourget-du-Lac, France. E-mail: tom.hirschowitz@univ-savoie.fr

Xavier Leroy
INRIA Paris-Rocquencourt, Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France. E-mail: xavier.leroy@inria.fr

J. B. Wells
School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, Great Britain. E-mail: jbw@macs.hw.ac.uk

evaluation correctly reaches the fixed point when it exists, or diverges when the recursive definition is ill-founded, as in $x = x + 1$. 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, statically or dynamically.

The simplest way to rule out ill-founded definitions and ensure call-by-value evaluability is to syntactically restrict the right-hand sides of recursive definitions to be function abstractions, as ML does. Such a restriction also enables efficient compilation of the recursive definitions, for instance using the compilation scheme described by Appel [1]. While generally acceptable for direct programming in ML, this restriction can be problematic when we wish to encode higher-level constructs such as objects, classes, recursive modules and mixin modules. For instance, Boudol [3] uses definitions of the shape $x = c\ x$ (where c is a variable) in his recursive record semantics of objects. Similarly, Hirschowitz and Leroy [14] use mutually-dependent sets of such definitions for representing mixin modules. Putting these works into practice requires the definition of an efficient, call-by-value intermediate language supporting such non-standard recursive definitions. This definition is the topic of the present article.

1.2 From backpatching to immediate in-place update

Backpatching of reference cells A famous example of a call-by-value language that does not statically restrict the right-hand sides of recursive definitions is Scheme [17]. The operational semantics of the `letrec` construct of Scheme is known as the *backpatching* semantics[1]. It is illustrated in Figure 1. Consider two mutually-dependent definitions $x_1 = e_1$ and $x_2 = e_2$. First, a reference cell is assigned to each recursive variable, and initialized to some dummy value `undefined` (represented by $\bullet$ in Figure 1). Then, the right-hand sides are evaluated, building data structures that possibly include the reference cells, to obtain some values v_1 and v_2. Until this point, any attempt to dereference the cells is a run-time error. Finally, the reference cells are updated with v_1 and v_2, and the definitions can be considered fully evaluated.

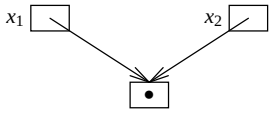
The backpatching scheme leaves some flexibility as to when the reference cells bound to recursively-defined variables are dereferenced. In Scheme, every occurrence of these variables that is evaluated in the lexical scope of the `letrec` binding causes an immediate dereference. Boudol and Zimmer [4] propose a compilation scheme for a call-by-value λ-calculus with unrestricted mutually recursive definitions where the dereferencing is further delayed because arguments to functions are passed by reference rather than by value. The difference is best illustrated on the definition $x = (\lambda y.\lambda z.\texttt{if}\ z = 0\ \texttt{then}\ 1\ \texttt{else}\ y\ (z-1))\ x$. In Scheme, it compiles down to the following intermediate code (written in ML-style notation)

$$\begin{array}{l}\texttt{let}\ x = \texttt{ref undefined in}\\ x := (\lambda y.\lambda z.\texttt{if}\ z = 0\ \texttt{then}\ 1\ \texttt{else}\ y\ (z-1))\ !x\end{array}$$

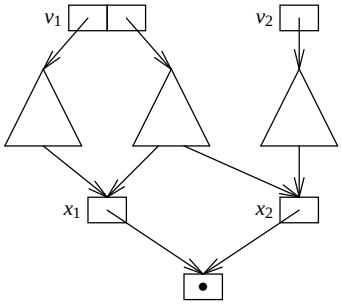
and therefore fails at run-time because the reference x is accessed at a time when it still contains `undefined`. In Boudol and Zimmer's compilation scheme, the y parameter is passed

[1] The *immediate in-place update* compilation scheme studied in this paper also uses a kind of backpatching, but we only use "backpatching" to refer to the schemes described in this section, i.e., to abbreviate "backpatching of reference cells".

1. Initialization:



2. Computation:



3. Reference update:

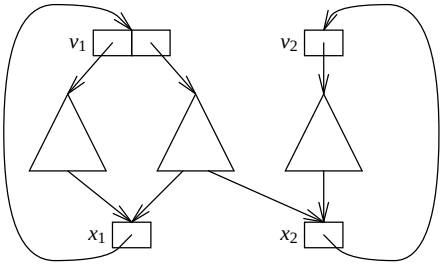


Fig. 1 The backpatching scheme

by reference, resulting in the following compiled code:

$$\texttt{let}\, x = \texttt{ref undefined in}$$
$$x := (\lambda y.\lambda z.\texttt{if}\ z = 0\ \texttt{then}\ 1\ \texttt{else}\ !y\,(z-1))\ x$$

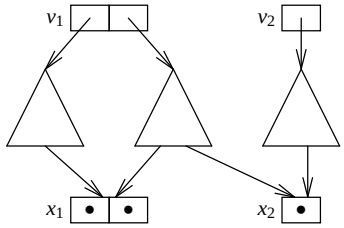
Here, x is passed as a function argument without being dereferenced, therefore ensuring that the recursive definition evaluates correctly. The downside is that the recursive call to y has now to be preceded by a dereferencing of y.

In summary, the backpatching semantics featured in Scheme enables a wider range of recursive definitions to be evaluated under a call-by-value regime than the syntactic restriction of ML. This range is even wider in Boudol and Zimmer's variant [4]. In both cases, a drawback of this approach is that, in general, recursive calls to a recursively-defined function must go through one additional indirection. For well-founded definitions, this indirection seems superfluous, since no further update of the reference cells is needed. Scheme compilers optimize this indirection away in some cases, typically when the right-hand sides are syntactic functions; but removing it in all cases requires alternative approaches, which we now describe.

In-place update The *in-place update* scheme [6] is a variant of the backpatching implementation of recursive definitions that avoids the additional indirection just mentioned. It is used in the OCaml compilers [21].

1. Pre-allocation:

2. Computation:



3. In-place update:

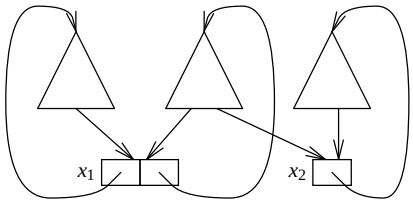


Fig. 2 The in-place update scheme

The in-place update scheme implements mutually recursive definitions that satisfy the following two conditions. For a mutually recursive definition $x_1 = e_1, \ldots, x_n = e_n$, first, the value of each definition should be represented at run-time by a heap allocated block of statically predictable size; second, for each i, the computation of e_i should not need the value of any of the definitions e_j, but only their names x_j. As an example of the second condition, the recursive definition $f = \lambda x.(\ldots f \ldots)$ is accepted, since the computation of the right-hand side does not need the value of f. We say that it *weakly* depends on f. In contrast, the recursive definition $f = (f\ 0)$ is rejected. We say that the right-hand side *strongly* depends on f. Several techniques to check this condition have been proposed [3, 14, 12, 8].

The evaluation of a set of mutually recursive definitions with in-place update consists of three steps. First, for each definition, allocate an uninitialized block of the expected size, and bind it to the recursively-defined identifier. Those blocks are called *dummy* blocks, and this step is called the *pre-allocation* step. 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, update the dummy blocks in place with the contents of the computed blocks. (Alternatively, the second step could store directly its results in the dummy blocks. However, this would require a special evaluation scheme for right-hand sides of recursive definitions whereas, here, they are evaluated just like any other expression.)

For example, consider a mutually recursive definition $x_1 = e_1, x_2 = e_2$, where it is statically predictable that the values of the expressions e_1 and e_2 will be represented at runtime by heap-allocated blocks of sizes 2 and 1, respectively. Here is what the compiled code does, as depicted in Figure 2. First, it allocates two uninitialized heap blocks, at addresses ℓ_1 and ℓ_2, of respective sizes 2 and 1. Then, it computes e_1, where x_1 and x_2 are bound to ℓ_1 and ℓ_2, respectively. The result is a heap block of size 2, possibly containing references to ℓ_1

and ℓ_2. The same process is carried on for e_2, resulting in a heap block of size 1. The third and final step copies the contents of the two obtained blocks to ℓ_1 and ℓ_2, respectively, then garbage-collects the useless blocks. The result is that the two initially dummy blocks now contain the proper cyclic data structures, without the indirection inherent in the backpatching semantics.

Immediate in-place update The scheme described above computes all definitions in sequence, and only then updates the dummy blocks in place. From the example above, it seems quite clear that in-place update for a definition could be done as soon as its value is available. Such an improvement has been proposed for the backpatching semantics [31], and we merely adapt it to our setting here. We call this method the *immediate in-place update* scheme and concentrate on it in the remainder of this paper.

As long as definitions weakly depend on each other, as happens with functions for instance, both schemes behave identically. Nevertheless, in the case where e_2 strongly depends on x_1, for example if e_2 = `fst`$(x_1)+1$, the original scheme can go wrong. Indeed, the contents of ℓ_1 are still undefined when e_2 is computed. Instead, with immediate in-place update, the value v_1 is already available when computing e_2. This trivial modification to the scheme thus increases the expressive power of mutually recursive definitions. It allows definitions to de-structure the values of previous definitions. Furthermore, it allows some of the mutually-recursive definitions to have statically unknown sizes, as shown by the following example.

An example of execution is presented in Figure 3. The definition is $x_1 = e_1, x_2 = e_2, x_3 = e_3$, where e_1 and e_3 are expected to evaluate to blocks of sizes 2 and 1, respectively, but where the representation for the value of e_2 is not statically predictable. The pre-allocation step allocates dummy blocks for x_1 and x_3 only. The value v_1 of e_1 is then computed. It can reference x_1 and x_3, which correspond to pointers to the dummy blocks, but not x_2, which would not make any sense here. This value is copied to the corresponding dummy block. Then, the value v_2 of e_2 is computed. The computation can refer to both dummy blocks, and can also strongly depend on x_1, but not on x_2. Finally, the value v_3 of e_3 is computed and copied to the corresponding dummy block.

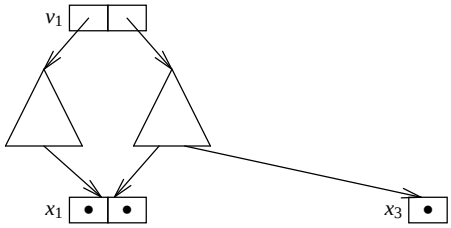
The immediate in-place update scheme implements more definitions than the original in-place update scheme. In fact, it implements arbitrary non-recursive definitions, thus allowing to merge the traditionally distinct constructs `let` and `let rec`.

Restrictions imposed on the source language What are the restrictions put on recursive definitions in the source language if we are to compile them with the immediate in-place update scheme? We adopt the following sufficient conditions. First, the values of forward referenced definitions must be represented by heap-allocated blocks. Second, the sizes of these blocks must be known statically. Third, the contents of these blocks should not be accessed before they have been updated with proper values. These restrictions are highly dependent on the data representation strategy implemented by the compiler. The second restriction also depends on how expected sizes are computed at compile-time, which entails a static analysis that is necessarily conservative. For instance, Hirschowitz [12] derives the sizes from the static types of the right-hand sides of recursive definitions, while the OCaml compiler proceeds by syntactic inspection of the shapes of the right-hand sides. More sophisticated static analyses, such as 0-CFA [29] or enriched type systems, could also be used.

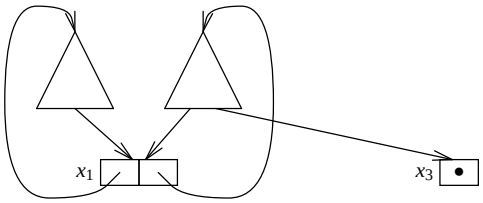
In this article, we abstract over these compiler-dependent issues as follows. We define a source language where each recursive definition is annotated by the expected size of the representation of the right-hand side, if known. These annotations reflect the result of a prior size analysis of the kind mentioned earlier. Both our source and target languages feature a

1. Pre-allocation:

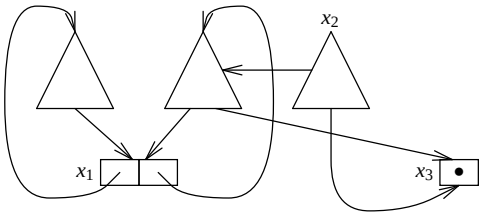
2. Computation of e_1:



3. Update of x_1 with v_1:



4. Computation of e_2 and binding of its value to x_2:



5. Computation and update of e_3:

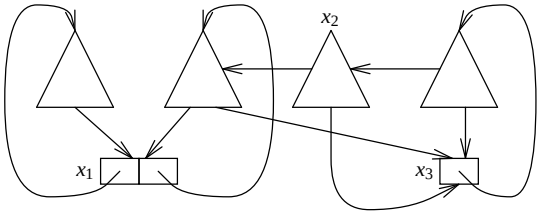


Fig. 3 The immediate in-place update scheme

notion of size, which we only assume to be preserved by the translation (Hypothesis 17) and satisfy a few natural requirements (Hypotheses 1 and 10).

1.3 Summary of contributions

The contributions of this article are threefold. First, we introduce and formalize a call-by-value functional language called $\lambda_\circ$, featuring an extended recursion construct that is not restricted to λ-abstractions as right-hand sides of recursive definitions, but also supports recursive definitions of data structures (x = `cons` 1 x) and of fixed points of certain higher-

order functions (`x = f x`). This recursion construct subsumes both the standard recursive and non-recursive value binding constructs `let` and `let rec`, and is compilable by immediate in-place update.

Second, we provide the first formalization of the in-place update implementation scheme. It is formalized as a translation from $\lambda_\circ$ to a target language λ_{a} that does not feature recursive definitions, but instead explicitly manipulates a heap via allocation and update operations. This language is designed to closely match the `Lambda` intermediate languages used by the OCaml compiler [21], attesting that it can be implemented efficiently.

Third, we prove that the evaluation of any $\lambda_\circ$ expression is correctly simulated by its translation. This is the first formal correctness proof for the in-place update scheme.

The remainder of this paper is organized as follows. In Section 2, we formalize the source language $\lambda_\circ$. Section 3 defines the target language λ_{a}. We define the compilation scheme from $\lambda_\circ$ to λ_{a} in Section 4, and prove its correctness in Section 5. Finally, we discuss related work in Section 6 and conclusions and future work in Section 7.

2 The source language $\lambda_\circ$

2.1 Notations

Given two sets A and B, $A \# B$ means that A and B are disjoint, $\mathscr{P}(A)$ denotes the set of all subsets of A, and $|A|$ denotes the cardinal of A.

For all sets A and B and functions $f : A \to B$, $\mathsf{dom}(f)$ denotes the *domain* A of f, and $\mathsf{cod}(f)$ denotes its *codomain* B. Moreover, $f_{\setminus C}$ denotes f restricted to $A \setminus C$. We also write $f\langle a \mapsto b\rangle$ for the unique function $f' : (A \cup \{a\}) \to (B \cup \{b\})$ such that $f'(a) = b$ and for all $a' \in A \setminus \{a\}$, $f'(a') = f(a')$. Moreover, for all functions $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$, if $A_1 \# A_2$, then $f_1 + f_2$ denotes the union of f_1 and f_2 as graphs.

For any syntactic entity ranged over by a meta variable X, with variables ranged over by x, the notation $[x_1 \mapsto X_1, \ldots, x_n \mapsto X_n]$ (for $n \geq 1$) denotes a *substitution* function σ that maps x_i to X_i for $1 \leq i \leq n$, and maps all other variables to themselves. The identity substitution is written id. The application of a substitution to a syntactic entity with bindings must use standard techniques to avoid variable capture. The domain of this substitution is the set of all variables, and its *support* $\mathsf{supp}(\sigma)$ is $\{x \mid x \neq \sigma(x)\}$. Substitutions are required to have finite support. Accordingly, the *cosupport* is defined by $\mathsf{cosupp}(\sigma) = \{\sigma(x) \mid x \in \mathsf{supp}(\sigma)\}$. For all substitutions σ_1 and σ_2, if $\mathsf{supp}(\sigma_1) \# \mathsf{supp}(\sigma_2)$, we define their disjoint union $\sigma_1 + \sigma_2$ by $(\sigma_1 + \sigma_2)(x) = \sigma_1(x)$ for all $x \in \mathsf{supp}(\sigma_1)$, $(\sigma_1 + \sigma_2)(x) = \sigma_2(x)$ for all $x \in \mathsf{supp}(\sigma_2)$, and $(\sigma_1 + \sigma_2)(x) = x$ for all $x \notin (\mathsf{supp}(\sigma_1) \uplus \mathsf{supp}(\sigma_2))$. (This overloads the previous notation $f_1 + f_2$ for functions with disjoint domains.) For all substitutions σ_1 and σ_2, we write $\sigma_1(\sigma_2)$ for the unique substitution of support $\mathsf{supp}(\sigma_2)$ such that for all $x \in \mathsf{supp}(\sigma_2)$, $\sigma_1(\sigma_2)(x) = \sigma_1(\sigma_2(x))$. It is in general different from the composition $\sigma_1 \circ \sigma_2$, since if $x \in \mathsf{supp}(\sigma_1) \setminus \mathsf{supp}(\sigma_2)$, $(\sigma_1 \circ \sigma_2)(x) = \sigma_1(x)$, whereas $(\sigma_1(\sigma_2))(x) = x$.

2.2 Syntax

The syntax of $\lambda_\circ$ is defined in Figure 4. The meta-variables X and x range over names and variables, respectively. Variables are used in binders, as usual. Names are used for labeling record fields. The metavariables for other syntactic entities are in lowercase, in order to ease the distinction with the metavariables for syntactic entities of the target language (Section 3),

Variable:		x	$\in$	vars
Name:		X	$\in$	names
Expression:	$e \in \mathsf{expr}$	$::=$	$x \mid \lambda x.e \mid e_1\ e_2$	λ-calculus
		$\mid$	$\{r\} \mid e.X$	Record operations
		$\mid$	$\mathtt{rec}\ b\ \mathtt{in}\ e$	Recursive definitions
Record row:		r	$::= \varepsilon \mid X = x, r$	
Binding:		b	$::= \varepsilon \mid x \diamond e, b$	
Size indication:		$\diamond$	$::= =_{[n]} \mid =_{[?]}$	(n a natural number)

Fig. 4 Syntax of $\lambda_\circ$

$$
\begin{array}{llll}
\mathsf{FV}(x) & = \{x\} & \mathsf{FV}(\lambda x.e) & = \mathsf{FV}(e) \setminus \{x\} \\
\mathsf{FV}(e_1\ e_2) & = \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) & \mathsf{FV}(\{r\}) & = \mathsf{FV}(r) \\
\mathsf{FV}(e.X) & = \mathsf{FV}(e) & \mathsf{FV}(\mathtt{rec}\ b\ \mathtt{in}\ e) & = (\mathsf{FV}(b) \cup \mathsf{FV}(e)) \setminus \mathsf{dom}(b) \\
\mathsf{FV}(b) & = \bigcup_{(x \diamond e) \in b} \{x\} \cup \mathsf{FV}(e) & \mathsf{FV}(r) & = \{r(X) \mid X \in \mathsf{dom}(r)\}
\end{array}
$$

Fig. 5 Free variables in $\lambda_\circ$

which will be in upper case. The syntax includes the λ-calculus: variable x, abstraction $\lambda x.e$, and application $e_1\ e_2$. The language also features records, record selection $e.X$ and a binding construct written `rec`. By convention, the `rec` construct has lowest precedence, so that for instance $\mathtt{rec}\ b\ \mathtt{in}\ e_1\ e_2$ means $\mathtt{rec}\ b\ \mathtt{in}\ (e_1\ e_2)$. In a $\mathtt{rec}\ b\ \mathtt{in}\ e$ expression, e is called the *body*. To simplify the formalization and without loss of expressiveness, records are restricted to contain only variables, i.e., be of the shape $\{X_1 = x_1, \ldots, X_n = x_n\}$. *Bindings* b have the shape $x_1 \diamond_1 e_1, \ldots, x_n \diamond_n e_n$, where arbitrary expressions are syntactically allowed as the right-hand sides of definitions, and every definition is annotated with a *size indication* $\diamond$. A size indication can be either the unknown size indication $=_{[?]}$, or a known size indication $=_{[n]}$, where n is a natural number. We write ε for the empty binding.

Implicit syntactic constraints In what follows, we implicitly restrict ourselves to record rows, bindings and expressions satisfying the following conditions:

1. Record rows do not define the same name twice;
2. Bindings do not define the same variable twice;
3. Bindings do not contain *forward references* to definitions of unknown size, in the sense made precise next.

The *free variables* $\mathsf{FV}(e)$ of expressions, bindings, and record rows are defined inductively by the rules in Figure 5. In a `rec` binding $b = (x_1 \diamond_1 e_1, \ldots, x_n \diamond_n e_n)$, we say that there is a *forward reference* of x_i to x_j if $i \leq j$ and $x_j \in \mathsf{FV}(e_i)$. Condition 3 requires that for all bindings b and forward reference of x_i to x_j in b, the size indication $\diamond_j$ is $=_{[n]}$ for some n. This is consistent with the immediate in-place update scheme, where no blocks are pre-allocated for definitions of unknown size, so previous definitions must not refer to them.

Finally, taking advantage of conditions 1 and 2 above, we implicitly view record rows as finite functions from names to variables and bindings as finite functions from variables to expressions, and use standard notations for domain, codomain, application, etc. Also, we write r_1, r_2 for the concatenation of r_1 and r_2, and similarly for bindings. Finally, we implicitly view records and bindings as sets of pairs (X, x) (resp. of triples $(x, \diamond, e)$), for example to write $(X = x) \in r$ (resp. $(x \diamond e) \in b$).

Value:	$v \in \mathsf{values}$	$::= x \mid \lambda x.e \mid \{r\}$	
Answer:	$a \in \mathsf{answers}$	$::= v \mid \mathtt{rec}\ b_\mathsf{v}\ \mathtt{in}\ v$	
Size-respecting binding:	b_v	$::= \varepsilon$	
		$\mid\ x =_{[?]} v, b_\mathsf{v}$	
		$\mid\ x =_{[n]} v, b_\mathsf{v}$	where $\mathsf{size}(v) = n$

Fig. 6 Values and answers in $\lambda_\circ$

Structural equivalence We consider expressions equivalent up to α-conversion[2], i.e., renaming of bound variables, in functions and `rec` expressions. In the following, to avoid ambiguity, we call *raw* expressions not considered up to α-conversion. Let $=$ denote equality of raw expressions and $\equiv$ denote equality modulo α conversion.

2.3 Dynamic semantics

We now define the dynamic semantics of $\lambda_\circ$. Figure 6 defines $\lambda_\circ$ values to be variables, functions, and records.

2.3.1 Overview: sizes and recursive definitions

We have seen that `rec`-bound definitions can be annotated with natural numbers representing their sizes. The role of these size indications is to declare in advance the expected sizes of the memory blocks representing the values of definitions. Technically, they will be required to match the size of allocated blocks in the sense of our target calculus. For definitions that are not forward-referenced from previous definitions, there is no need for annotations.

In $\lambda_\circ$, during the evaluation of a binding, if the currently evaluated definition is expected to have size n, then it must evaluate to a non-variable value whose size equals n. Otherwise, evaluation gets stuck.

Hypothesis 1 (Size in $\lambda_\circ$) We assume given a partial function size from $\lambda_\circ$ values to natural numbers, defined exactly on $\mathsf{values} \setminus \mathsf{vars}$.

An evaluated definition not matching its size indication is considered an error, in the sense that it prevents further reductions. Thus, only *size-respecting* bindings b_v, as defined in Figure 6, are considered fully evaluated.

Note that size-respecting bindings define only values. The intuition is that, given a definition $(x =_{[n]} e)$, this forces the topmost block of the value of e to be determined by previous definitions. For instance, suppose that $\mathsf{size}(\{X = x\}) = n$. Then, the binding $(y =_{[n]} \{X = x\}, z =_{[n]} y)$ is not fully evaluated, but we will see below that it evaluates correctly to $(y =_{[n]} \{X = x\}, z =_{[n]} \{X = x\})$. On the contrary, the binding $(z =_{[n]} y, y =_{[n]} \{X = x\})$ is invalid: y can not be replaced with its value, according to the reduction relation defined below. (Such a reduction step could not be implemented by immediate in-place update as depicted in Figure 3.)

Besides the non-standard notion of size, the dynamic semantics of $\lambda_\circ$ is unusual in its handling of mutually recursive definitions, which is adapted from the equational theory of

[2] The notion of structural equivalence could include reordering of record fields, but we do not need it, so we just consider α-equivalence.

Ariola and Blom [2]. There is no rule for eliminating `rec`: evaluated bindings remain at top-level in the expression and also in evaluation *answers*, as defined in Figure 6. This top-level binding serves as a kind of heap or recursive evaluation environment. An answer a is defined to be a value, possibly surrounded by an evaluated, size-respecting binding. It thus may have the shape $\texttt{rec}\ b_v\ \texttt{in}\ v$.

The dynamic semantics of `rec` relies on five fundamental equations, which resemble the rules used by Wright and Felleisen [32]. We start with an informal presentation of these equations using *contexts* $\mathbb{C}$, i.e., terms with a hole $\Box$. Context application $\mathbb{C}[e]$ is textual, possibly capturing replacement of $\Box$ with e in $\mathbb{C}$. The rules rely on additional conditions defined later to (1) avoid variable captures and (2) enforce the reduction strategy of the language, but are roughly as follows.

1. The first equation is *lifting*. It lifts a `rec` node up one level in an expression. An expression of the shape $e_1\ (\texttt{rec}\ b\ \texttt{in}\ e_2)$ is equated with $\texttt{rec}\ b\ \texttt{in}\ (e_1\ e_2)$.
2. The second equation is *internal merging*. In a binding, when one of the definitions starts with another binding, then this binding can be merged with the enclosing one. An expression of the shape $\texttt{rec}\ b_1, x = (\texttt{rec}\ b_2\ \texttt{in}\ e_1), b_3\ \texttt{in}\ e_2$ is equated with $\texttt{rec}\ b_1, b_2, x = e_1, b_3\ \texttt{in}\ e_2$.
3. The third equation is *external merging*, which merges two consecutive bindings. An expression of the shape $\texttt{rec}\ b_1\ \texttt{in}\ \texttt{rec}\ b_2\ \texttt{in}\ e$ is equated with $\texttt{rec}\ b_1, b_2\ \texttt{in}\ e$.
4. The fourth equation, *external substitution*, replaces variables defined in an enclosing binding with their definitions. Given a context $\mathbb{C}$, an expression of the shape $\texttt{rec}\ b\ \texttt{in}\ \mathbb{C}[x]$ is equated with $\texttt{rec}\ b\ \texttt{in}\ \mathbb{C}[e]$, if $x = e$ appears in b.
5. The last equation, *internal substitution*, replaces variables defined in the same binding with their definitions. Given a context $\mathbb{C}$, an expression of the shape $\texttt{rec}\ b_1, y = \mathbb{C}[x], b_2\ \texttt{in}\ e_1$ is equated with $\texttt{rec}\ b_1, y = \mathbb{C}[e_2], b_2\ \texttt{in}\ e_1$ if $x = e_2$ appears in $b_1, y = \mathbb{C}[x], b_2$.

The issue is how to arrange these operations to make the evaluation deterministic and to ensure that it reaches the answer when it exists. Our choice can be summarized as follows. First, bindings that are not at top-level in the expression must be lifted before their evaluation can begin. Thus, only the top-level binding can be evaluated. As soon as one of its definitions gets evaluated, evaluation can proceed with the next one, or with the body if there is no definition left. If evaluation encounters a binding inside the considered expression, then this binding is lifted up to the top level of the expression, or just before the top-level binding if there is one. In this case, it is merged with the latter, internally or externally, according to the context. External substitution is used to replace a variable in *dereferencing* position (like x in $x.X$ or $x\ v$, see the precise definition of dereferencing contexts below) with its value, fetched from the top-level binding. Internal substitution is used similarly, but inside the top-level binding, and only from left to right (i.e., when the copied definition comes from the left of the current evaluation point).

Remark 2 (Policy on substitution and call-by-value) The substitution rules only replace one occurrence of a variable at a time, which has to be in destructive position. This strategy w.r.t. substitution, called *destruct-time* by Sewell et al [28], does not contradict the fact that $\lambda_\circ$ is call-by-value. Indeed, only values are copied, and any expression reached by the evaluation is immediately evaluated. The fact that evaluated definitions are not immediately substituted with their values in the rest of the expression is rather a matter of presentation. Notably, this presentation allows $\lambda_\circ$ to properly represent recursive data structures, as shown in Section 2.4 and Figure 14.

To implement our strategy, we remark that evaluation should not be the same at top-level and inside an evaluation context. For example, consider $e \equiv ((\texttt{rec}\ x =_{[?]}\ e_0\ \texttt{in}\ x\ y)\ z)$,

Lift context:
$\mathbb{L} ::= e\ \square \mid \square\ v \mid \square.X$
Nested lift context:
$\mathbb{F} ::= \square \mid e\ \mathbb{F} \mid \mathbb{F}\ v \mid \mathbb{F}.X$
Evaluation context:
$\mathbb{E} ::= \mathbb{F}$
$\mid \mathtt{rec}\ b_v\ \mathtt{in}\ \mathbb{F}$
$\mid \mathtt{rec}\ b_v, x \diamond \mathbb{F}, b\ \mathtt{in}\ e$

Binding context:
$\mathbb{B}_\diamond ::= b_v, x \diamond \square, b$
Nested dereferencing context:
$\mathbb{A} ::= \square\ v \mid \square.X$
$\mid e\ \mathbb{A} \mid \mathbb{A}\ v \mid \mathbb{A}.X$
Dereferencing context:
$\mathbb{D} ::= \mathbb{A}$
$\mid \mathtt{rec}\ b_v\ \mathtt{in}\ \mathbb{A}$
$\mid \mathtt{rec}\ b_v, x \diamond \mathbb{A}, b\ \mathtt{in}\ e$
$\mid \mathtt{rec}\ \mathbb{B}_{=_{[n]}}\ \mathtt{in}\ e$

Fig. 7 Evaluation contexts of $\lambda_\diamond$

Alpha equivalence:

$$\frac{e \equiv e'}{e\ \mathbb{F} \equiv e'\ \mathbb{F}} \qquad \frac{v \equiv v'}{\mathbb{F}\ v \equiv \mathbb{F}\ v'} \qquad \frac{b_v \equiv b_v'}{\mathtt{rec}\ b_v\ \mathtt{in}\ \mathbb{F} \equiv \mathtt{rec}\ b_v'\ \mathtt{in}\ \mathbb{F}}$$

$$\frac{b_v \equiv b_v' \qquad b \equiv b' \qquad e \equiv e'}{(\mathtt{rec}\ b_v, x \diamond \mathbb{F}, b\ \mathtt{in}\ e) \equiv (\mathtt{rec}\ b_v', x \diamond \mathbb{F}, b'\ \mathtt{in}\ e')} \qquad \frac{\mathbb{F} \equiv \mathbb{F}'}{\mathbb{E}[\mathbb{F}] \equiv \mathbb{E}[\mathbb{F}']} \qquad \frac{e \equiv e'}{(b_1, x \diamond e, b_2) \equiv (b_1, x \diamond e', b_2)}$$

Free variables:
$\mathsf{FV}(\square) = \emptyset$
$\mathsf{FV}(e\ \mathbb{F}) = \mathsf{FV}(e) \cup \mathsf{FV}(\mathbb{F})$
$\mathsf{FV}(\mathbb{F}\ v) = \mathsf{FV}(\mathbb{F}) \cup \mathsf{FV}(v)$
$\mathsf{FV}(\mathbb{F}.X) = \mathsf{FV}(\mathbb{F})$
$\mathsf{FV}(\mathtt{rec}\ b_v\ \mathtt{in}\ \mathbb{F}) = \mathsf{FV}(b_v) \cup \mathsf{FV}(\mathbb{F})$
$\mathsf{FV}(\mathtt{rec}\ b_v, x \diamond \mathbb{F}, b\ \mathtt{in}\ e) = \{x\} \cup \mathsf{FV}(b_v, b) \cup \mathsf{FV}(\mathbb{F}) \cup \mathsf{FV}(e)$

Captured variables:
$\mathsf{Capt}_\square(\mathtt{rec}\ b_v\ \mathtt{in}\ \mathbb{F}) = \mathsf{dom}(b_v)$
$\mathsf{Capt}_\square(\mathtt{rec}\ b_v, x \diamond \mathbb{F}, b\ \mathtt{in}\ e) = \{x\} \cup \mathsf{dom}(b_v, b)$
$\mathsf{Capt}_\square(\mathbb{F}) = \emptyset$

Fig. 8 Structural equivalence of $\lambda_\diamond$ evaluation contexts

where e_0 reduces to e_1. According to the informal specification above, before the evaluation of e_0 can start, the binding should first be lifted to the top level to obtain $e' \equiv (\mathtt{rec}\ x =_{[?]} e_0\ \mathtt{in}\ (x\ y\ z))$. So, our reduction relation should not respect the usual rule saying that for any e_0 and e_1, if $e_0 \longrightarrow e_1$, then $\mathbb{E}[e_0] \longrightarrow \mathbb{E}[e_1]$ for any evaluation context $\mathbb{E}$. This leads us to define two relations: the *subreduction relation* $\rightsquigarrow$, handling reductions inside expressions, and the *reduction relation* $\longrightarrow$, handling top-level reductions. We write $\rightsquigarrow^+$ (resp. $\rightsquigarrow^*$) for the transitive (resp. transitive reflexive) closure of the relation $\rightsquigarrow$, and similarly for $\longrightarrow$.

2.3.2 The subreduction relation

First, we define subreduction in Figure 9, using notions defined in Figures 7 and 8. It is first defined on raw expressions, then lifted to α-equivalence classes of expressions by the usual rule

$$\frac{e_1 \equiv e_1' \qquad e_1' \rightsquigarrow e_2' \qquad e_2' \equiv e_2}{e_1 \rightsquigarrow e_2}$$

Record projection selects the appropriate field in the record (rule PROJECT$_\diamond$). The application of a function $\lambda x.e$ to a value v reduces to the body of the function where the argument has been `rec`-bound to x (rule BETA$_\diamond$). Rule LIFT$_\diamond$ describes how bindings are

Subreduction rules ($\rightsquigarrow$)

$$\{r\}.X \rightsquigarrow r(X) \quad (\textsc{Project}_\circ) \qquad \frac{x \notin \mathsf{FV}(v)}{(\lambda x.e)\ v \rightsquigarrow \mathtt{rec}\ x =_{[?]} v\ \mathtt{in}\ e} \quad (\textsc{Beta}_\circ)$$

$$\frac{\mathsf{dom}(b) \# \mathsf{FV}(\mathbb{L})}{\mathbb{L}[\mathtt{rec}\ b\ \mathtt{in}\ e] \rightsquigarrow \mathtt{rec}\ b\ \mathtt{in}\ \mathbb{L}[e]} \quad (\textsc{Lift}_\circ)$$

Reduction rules ($\longrightarrow$)

$$\frac{e \rightsquigarrow e'}{\mathbb{E}[e] \longrightarrow \mathbb{E}[e']} \quad (\textsc{Context}_\circ)$$

$$\frac{\mathsf{dom}(b_1) \# (\{x\} \cup \mathsf{FV}(b_\mathsf{v}, b_2) \cup \mathsf{FV}(e'))}{(\mathtt{rec}\ b_\mathsf{v}, x \diamond (\mathtt{rec}\ b_1\ \mathtt{in}\ e), b_2\ \mathtt{in}\ e') \longrightarrow (\mathtt{rec}\ b_\mathsf{v}, b_1, x \diamond e, b_2\ \mathtt{in}\ e')} \quad (\mathrm{IM}_\circ)$$

$$\frac{\mathsf{dom}(b) \# \mathsf{FV}(b_\mathsf{v})}{(\mathtt{rec}\ b_\mathsf{v}\ \mathtt{in}\ \mathtt{rec}\ b\ \mathtt{in}\ e) \longrightarrow \mathtt{rec}\ b_\mathsf{v}, b\ \mathtt{in}\ e} \quad (\mathrm{EM}_\circ) \qquad \mathbb{D}[x] \longrightarrow \mathbb{D}[\mathbb{D}(x)] \quad (\textsc{Subst}_\circ)$$

Fig. 9 Dynamic semantics of $\lambda_\circ$

lifted up to the top of the term. *Lift contexts* $\mathbb{L}$ are defined in Figure 7. Rule $\textsc{Lift}_\circ$ states that an expression of the shape $\mathbb{L}[\mathtt{rec}\ b\ \mathtt{in}\ e]$ subreduces to $\mathtt{rec}\ b\ \mathtt{in}\ \mathbb{L}[e]$, provided no variable capture occurs. Alpha-equivalence is defined over contexts as follows: all variables may be α-renamed, except those that have $\Box$ in their scope. More formally, α-equivalence for evaluation contexts is the smallest equivalence relation over evaluation contexts respecting the rules in Figure 8. In the same figure, we define the *captured* variables $\mathsf{Capt}_\Box(\mathbb{E})$ of an evaluation context $\mathbb{E}$, and the free variables of an evaluation context. We have $\mathsf{Capt}_\Box(\mathbb{E}) \subseteq \mathsf{FV}(\mathbb{E})$ for all $\mathbb{E}$.

Remark 3 (Evaluation order) Function applications are evaluated from right to left. This nonstandard choice is explained in Remark 11, in light of the semantics of the target language λ_a. The results of the paper can be adapted to a left-to-right evaluation setting with some additional work.

2.3.3 The reduction relation

The reduction relation is defined in Figure 9. It is first defined on raw expressions, then lifted to α-equivalence classes of expressions by the usual rule

$$\frac{e_1 \equiv e_1' \qquad e_1' \longrightarrow e_2' \qquad e_2' \equiv e_2}{e_1 \longrightarrow e_2}.$$

Rule $\textsc{Context}_\circ$ extends the subreduction relation (as a relation over raw expressions) to any evaluation context. As defined in Figure 7, we call a *nested lift context* $\mathbb{F}$ a series of lift contexts. Moreover, we call a *binding context* $\mathbb{B}_\diamond$ of size $\diamond$ a binding $(b_\mathsf{v}, x \diamond \Box, b)$ where the context hole $\Box$ corresponds to the next definition to be evaluated, and this definition is annotated by $\diamond$. An *evaluation context* $\mathbb{E}$ is a nested lift context, possibly appearing as the next definition to evaluate in the top-level binding, or enclosed inside a fully evaluated top-level binding. Our unusual, staged formulation of evaluation contexts enforces the determinism of the reduction relation w.r.t. bindings: evaluation never takes place inside or after a binding, except the top-level one. Other bindings inside the expression first have to be lifted to

the top by rule LIFT$_\circ$, then be merged with the top-level binding, if any, by rules EM$_\circ$ and IM$_\circ$ (respectively for external and internal merging). If the top-level binding is of the shape $b_\mathsf{v}, x \diamond (\mathtt{rec}\ b_1\ \mathtt{in}\ e), b_2$, rule IM$_\circ$ allows to merge b_1 with it, obtaining $b_\mathsf{v}, b_1, x \diamond e, b_2$. When an inner binding has been lifted to the top level, if there is already a top-level binding, then the two bindings are merged together by rule EM$_\circ$. This implements the strategy informally described above.

Finally, rule SUBST$_\circ$ describe how the variables defined by the top-level binding are replaced with their values when needed, i.e., when they appear in a *dereferencing context*, as defined in Figure 7. Dereferencing contexts may take two forms. First, they can be binding contexts of known size $\mathtt{rec}\ b_\mathsf{v}, x =_{[n]} \Box, b\ \mathtt{in}\ e$. In the immediate in-place update compilation scheme, any definition of known size yields an allocation of a dummy block, which has to be updated. This is reflected here by requiring that in definitions of the shape $(x =_{[n]} y), y$ be eventually replaced with a non-variable value of size n. Dereferencing contexts can also be *nested dereferencing contexts*, i.e., function applications $\Box\ v$ or record field selection $\Box.X$, wrapped by an evaluation context, as defined in Figure 7. Therefore, in $\lambda_\circ$, the value of a variable is copied only when needed for function application or record selection (or in-place update, implicitly). The value of a variable x is found in the current evaluation context, as formalized by the following notion of access in evaluation contexts.

Definition 4 Define
$$\begin{aligned}
&\mathsf{Binding}(\mathbb{F}) && \equiv \varepsilon\\
&\mathsf{Binding}(\mathtt{rec}\ b_\mathsf{v}\ \mathtt{in}\ \mathbb{F}) && \equiv b_\mathsf{v}\\
&\mathsf{Binding}(\mathtt{rec}\ b_\mathsf{v}, x \diamond \mathbb{F}, b\ \mathtt{in}\ e) && \equiv b_\mathsf{v},
\end{aligned}$$

The value $\mathbb{E}(x)$ of x in $\mathbb{E}$ is $(\mathsf{Binding}(\mathbb{E}))(x)$, when the latter is defined.

Lemma 5 (Determinism of evaluation) *The $\longrightarrow$ relation is deterministic.*

Proof We prove the result for raw expressions first, and then extend it to α-equivalence classes. First, subreduction is obviously deterministic, on raw expressions as well as on α-equivalence classes. Furthermore, both on raw expressions and on α-equivalence classes, the reduction rules do not overlap, so we only have to prove that each rule is deterministic.

First consider the case of raw expressions. For all evaluation contexts $\mathbb{E}_1, \mathbb{E}_2$ and subreduction redexes e_1 and e_2, if $\mathbb{E}_1[e_1] = \mathbb{E}_2[e_2]$, we show that $\mathbb{E}_1 = \mathbb{E}_2$. This is shown in three steps: for lift contexts (by case analysis), nested lift contexts (by induction), and evaluation contexts (by case analysis). Hence, rule CONTEXT$_\circ$ is deterministic. Similarly, rule SUBST$_\circ$ is deterministic.

Consider now the case of α-equivalence classes. Let a *renaming* ρ be a substitution function (as defined in Section 2.1) from variables to variables, and let $\rho(e)$ denote capture-avoiding substitution in the usual sense. For all evaluation contexts $\mathbb{E}_1, \mathbb{E}_2$ and sub reduction redexes e_1 and e_2, if $\mathbb{E}_1[e_1] \equiv \mathbb{E}_2[e_2]$, then there exists a renaming ρ, such that $\mathsf{supp}(\rho) \subseteq \mathsf{Capt}_\Box(\mathbb{E}_1)$ and $\rho(\mathbb{E}_1) \equiv \mathbb{E}_2$ and $\rho(e_1) \equiv e_2$. This entails that rule CONTEXT$_\circ$ is deterministic. We proceed similarly for rule SUBST$_\circ$. □

Definition 6 (Faulty $\lambda_\circ$ expression) A *faulty* $\lambda_\circ$ expression is an expression whose reduction gets stuck on an expression that is not an answer. By determinism, a non-faulty expression is an expression whose evaluation either does not terminate or reaches an answer.

We now characterize faulty expressions, using the following notion of *decomposition* of an expression e: a *decomposition* of an expression e is a pair $(\mathbb{E}, e')$ such that $e \equiv \mathbb{E}[e']$. (We consider pairs $(\mathbb{E}, e)$ modulo renaming of the captured variables of $\mathbb{E}$, hence decomposition is well-defined on α-equivalence classes of expressions.)

Let us now define an ordering over decompositions. Decompositions $(\mathbb{E}, e')$ induce occurrences in the abstract syntax tree of expressions, i.e., paths from its root to the designated occurrence of e'. This assignment is injective, i.e., these paths characterize decompositions. However, it is not onto since some paths do not correspond to any evaluation context. Given two decompositions $(\mathbb{E}, e)$ and $(\mathbb{E}', e')$ of some given e_0, corresponding to paths p and p', consider their maximal common prefix p''. We say that $p \sqsubseteq p'$ when either:

- $p'' = p$, or
- p', after p'', turns left in an application, i.e., p'' corresponds to a decomposition $(\mathbb{E}'', (\mathbb{F}[e]\ v))$ and $\mathbb{E}' \equiv \mathbb{E}''[\mathbb{F}\ v]$ (the other decomposition thus has $\mathbb{E} \equiv \mathbb{E}''[\mathbb{F}[e]\ \Box]$), or
- p', after p'', goes further in the top-level binding than p, i.e., $\mathbb{E} \equiv (\mathtt{rec}\ b_\mathsf{v}, x \diamond \Box, b\ \mathtt{in}\ e_1)$, e is a value (of the expected size if needed), and $\mathbb{E}'$ has shape $\mathtt{rec}\ b_\mathsf{v}, x \diamond v, b_\mathsf{v}', y \diamond \mathbb{F}, b'\ \mathtt{in}\ e_1$ or $\mathtt{rec}\ b_\mathsf{v}, b_\mathsf{v}'\ \mathtt{in}\ \mathbb{F}$.

This relation $\sqsubseteq$ defines a total ordering on the set of decompositions of any expression e, which furthermore has a maximal element – the decomposition turning left in applications when possible, and going as far as possible in the top-level binding. Using this notion, we prove the following characterization.

Proposition 7 *For all e, the following are equivalent:*

1. *e is faulty;*
2. *e reduces to an expression $\mathbb{D}[v]$ in normal form, such that if $\mathbb{D} \equiv \mathtt{rec}\ \mathbb{B}_{=_{[n]}}\ \mathtt{in}\ e$ for some n, $\mathbb{B}_{=_{[n]}}$, and e, then $\mathsf{size}(v)$, if defined, is not n;*
3. *e reduces to an expression e_0 such that:*
   - *$e_0 \equiv \mathbb{D}[x]$, with $\mathbb{D}(x)$ undefined,*
   - *$e_0 \equiv \mathtt{rec}\ b_\mathsf{v}, x =_{[n]} v, b\ \mathtt{in}\ e'$ with $\mathsf{size}(v) \neq n$ and $v \notin \mathsf{vars}$,*
   - *$e_0 \equiv \mathbb{E}[\{r\}\ v]$,*
   - *$e_0 \equiv \mathbb{E}[\{r\}.X]$ with $X \notin \mathsf{dom}(r)$,*
   - *$e_0 \equiv \mathbb{E}[(\lambda x.e').X]$.*

Moreover, for all x and $\mathbb{D}$, $\mathbb{D}(x)$ is undefined if and only if

- *either $x \notin \mathsf{Capt}_\Box(\mathbb{D})$,*
- *or $\mathbb{D} \equiv \mathtt{rec}\ b_\mathsf{v}, x' \diamond \mathbb{F}, b\ \mathtt{in}\ e'$, with $x \in \{x'\} \cup \mathsf{dom}(b)$.*

Proof First, observe that all cases of (3) are faulty, hence (3) implies (1). We now show that (1) implies (3).

Consider an expression with a normal form e which is not an answer. Consider its maximal decomposition $(\mathbb{E}, e')$ w.r.t. the ordering $\sqsubseteq$. The expression e is an answer exactly when e' is a value and $\mathbb{E}$ is either empty or of the shape $\mathtt{rec}\ b_\mathsf{v}\ \mathtt{in}\ \Box$. We proceed by case analysis on the other cases.

If e' is not a value, then by maximality, it has the shape $\mathtt{rec}\ b'\ \mathtt{in}\ e''$ for some b' and e'', and $\mathbb{E}$ is not empty. But then one of rules $\mathrm{IM}_\circ$, $\mathrm{EM}_\circ$, and $\mathrm{LIFT}_\circ$ applies, contradicting the fact that e is in normal form.

If e' is a value v, then $\mathbb{E}$ must have shape $\mathtt{rec}\ b_\mathsf{v}\ \mathtt{in}\ \mathbb{F}$ or $\mathtt{rec}\ b_\mathsf{v}, x \diamond \mathbb{F}, b\ \mathtt{in}\ e''$.

If $\mathbb{F}$ is not empty, then $\mathbb{E}$ has the shape $\mathbb{E}'[\mathbb{L}]$. Now, if $\mathbb{L} \equiv (e\ \Box)$, the decomposition $(\mathbb{E}, e')$ cannot be maximal, since the decomposition $(\mathbb{E}'[\Box\ v], e)$ is greater. Otherwise, if $\mathbb{L} \equiv (\Box\ v')$, then we have $\mathbb{E}'[v\ v']$ in normal form, hence either v is a variable undefined in $\mathbb{E}'$, or is a record. Otherwise, $\mathbb{L} \equiv (\Box.X)$, hence either v is a variable undefined in $\mathbb{E}'$, or is a function, or is a record without an X field. All these cases are covered by (3).

If otherwise $\mathbb{F}$ is empty, then $\mathbb{E}$ must have the shape $\mathtt{rec}\ b_\mathsf{v}, x \diamond \Box, b\ \mathtt{in}\ e''$. But then, for the decomposition $(\mathbb{E}, e')$ to be maximal, we must have $\diamond = =_{[n]}$ for some n, and either

– v is a variable undefined in $\mathbb{E}$ (first case of (3)), or
– $\mathsf{size}(v)$ is defined and different from n (second case).

Finally, to show the equivalence with (2), all the cases of (3) are covered by (2), so (3) implies (2), and the only possibility for an expression $\mathbb{D}[v]$ in normal form to be an answer is that $\mathbb{D}$ has the shape $\mathtt{rec}\ \mathbb{B}_{=_{[n]}}\ \mathtt{in}\ e$ with $\mathsf{size}(v) = n$, so (2) implies (1). □

Remark 8 In $\lambda_\circ$, we restrict record values to contain only variables. Actually, we could permit other kinds of values in record expressions, but not in record values, because it would break the properties of $\lambda_\circ$ w.r.t. sharing. In particular, as we also mention in Section 2.4, the sharing properties of $\lambda_\circ$ make it directly extensible with mutable values. If we allowed non-variable values in record values, then this would no longer be the case.

To see this, assume that $\lambda_\circ$ is extended with such record values and a ternary operator $e.X \leftarrow e'$ for mutation of record fields. Then, consider $e \equiv (\mathtt{rec}\ x =_{[?]} \{X = \{Y = v\}\}\ \mathtt{in}\ x.X.Y \leftarrow v')$. The evaluation of e is as follows: first, the record is copied, then its X field is projected, which gives $\mathtt{rec}\ x =_{[?]} \{X = \{Y = v\}\}\ \mathtt{in}\ \{Y = v\}.Y \leftarrow v'$, which is impossible to rewrite to the expected result.

In addition to this undesirable behavior, enriching $\lambda_\circ$ with non-variable values in record values would force us to considerably enrich the equational theory of our target language $\lambda_{\mathtt{a}}$. Indeed, $\lambda_{\mathtt{a}}$ gives a rather fine-grained account of sharing, and we would have to add equations to reason modulo sharing.

2.4 Examples

In this section, we show examples of $\lambda_\circ$ reduction and give intuitions on important applications of $\lambda_\circ$, namely mixin modules and recursive modules. These examples demonstrate the expressive power of $\lambda_\circ$, compared to the recursion constructs of both ML and Scheme, and also compared to the conference version of this paper [15]. Other possible applications include encodings of objects following Boudol [3]. However, $\lambda_\circ$ would have to be (straightforwardly) extended with mutable records to support this encoding.

2.4.1 Basic examples

We start with small examples to give some intuition on the semantics. First, as noted in Remark 2, substitution occurs at destruct-time in $\lambda_\circ$, following the terminology of [28]. This means that substitution of an occurrence of a variable is only performed when this occurrence has to be replaced with a non-variable value in order for the evaluation to continue. This is illustrated in Figure 10, which shows an example of substitution at function application time. The first expression is partitioned into $\mathbb{D} \equiv \mathtt{rec}\ x =_{[?]} \lambda y.y\ \mathtt{in}\ \Box\ x$ and x.

Figure 11 illustrates the left-to-right evaluation of bindings in $\lambda_\circ$ and the semantics of size indications. In particular, it emphasizes the fact that if a size indication turns out to be wrong, then the reduction is stuck. With respect to compilation, this models the fact that in the in-place update method, pre-allocated blocks should not be updated with larger blocks, otherwise execution might go wrong. In the second example of Figure 11, whose evaluation is correct, the first expression is partitioned into $\mathbb{D} \equiv \mathtt{rec}\ x =_{[n]} \lambda y.y, z =_{[?]} \Box\ x\ \mathtt{in}\ z$ and x.

Figure 12 shows a subtle point of the semantics. Namely, the size indications change the degree of sharing of definitions, in case they are just variables. From Figure 7, we remark that a binding context of the shape $x =_{[n]} \Box$ is dereferencing. Therefore, if it is filled with a

Expression	Comments
$\texttt{rec}\ x =_{[?]} \lambda y.y\ \texttt{in}\ x\ x$	Not a valid answer because, $x\ x$ is not a value, and the only possible reduction is by rule SUBST$_\circ$.
$\downarrow$	
$\texttt{rec}\ x =_{[?]} \lambda y.y\ \texttt{in}\ (\lambda y.y)\ x$	Only the first occurrence of x is substituted. We then apply rule BETA$_\circ$.
$\downarrow$	
$\texttt{rec}\ x =_{[?]} \lambda y.y\ \texttt{in}$ $\texttt{rec}\ y =_{[?]} x\ \texttt{in}\ y$	Not yet a valid answer. We apply rule EM$_\circ$.
$\downarrow$	
$\texttt{rec}\ x =_{[?]} \lambda y.y, y =_{[?]} x\ \texttt{in}\ y$	The binding is now size-respecting, because of the $=_{[?]}$.

Fig. 10 Substitution and function application

Expression	Comments
$\texttt{rec}\ z =_{[?]} x\ x,$ $\quad x =_{[n]} \lambda y.y$ $\texttt{in}\ z$ $\not\downarrow$	The forward reference is syntactically correct (even if $n \neq \mathsf{size}(\lambda y.y)$), but the value of x cannot be copied, because it would be from right to left. This is consistent with the in-place update compilation scheme sketched in Section 1.2.
$\texttt{rec}\ x =_{[n]} \lambda y.y,$ $\quad z =_{[?]} x\ x$ $\texttt{in}\ z$ $\downarrow$ $\texttt{rec}\ x =_{[n]} \lambda y.y,$ $\quad z =_{[?]} (\lambda y.y)\ x$ $\texttt{in}\ z$ $\vdots$	The value of x can be copied, but only if the size indication is correct, otherwise the first definition is not considered valid. Note that the size indication is in fact not necessary here because x is not forward referenced.

Fig. 11 Forward references

Expression	Comments
$\texttt{rec}\ y =_{[?]} \{X = \{\}\},$ $\quad z =_{[?]} y$ $\texttt{in}\ z$	The definition $z =_{[?]} y$ respects sizes, so the whole expression is an answer.
$\texttt{rec}\ y =_{[?]} \{X = \{\}\},$ $\quad z =_{[n]} y$ $\texttt{in}\ z$ $\downarrow$ $\texttt{rec}\ y =_{[?]} \{X = \{\}\},$ $\quad z =_{[n]} \{X = \{\}\}$ $\texttt{in}\ z$	The definition $z =_{[n]} y$ does not respect sizes, so the expression reduces by rule SUBST$_\circ$. We eventually reach an answer.

Fig. 12 Size indications and dereferencing contexts

Expression	Comments
rec *even* $=_{[?]} \lambda x.\ (x = 0)$ **or** $(odd\ (x-1))$, *odd* $=_{[n]} \lambda x.\ (x > 0)$ **and** $(even\ (x-1))$ **in** *even* 56	The forward reference to *odd* is syntactically correct, and *odd* evaluates correctly if *n* is the right size. We apply rule SUBST$_\circ$ to replace *even* with its definition.
$\downarrow$	
rec *even* $=_{[?]} \ldots$, *odd* $=_{[n]} \ldots$ **in** $(\lambda x.(x = 0)$ **or** $(odd\ (x-1)))\ 56$	We apply rule BETA$_\circ$, followed by rule EM$_\circ$.
$\downarrow_+$	
rec *even* $=_{[?]} \ldots$, *odd* $=_{[n]} \ldots$, $x_1 =_{[?]} 56$ **in** $(x_1 = 0)$ **or** $(odd\ (x_1 - 1))$	We then perform the boolean test unsuccessfully, obtaining $odd\ (x_1 - 1)$, where we then replace x_1 with its value and obtain *odd* 55. We can then replace *odd* with its value and apply rule BETA$_\circ$ again, and so on.
$\downarrow_+$	
rec *even* $=_{[?]} \ldots$, *odd* $=_{[n]} \ldots$, $x_1 =_{[?]} 56$ **in** *odd* 55	
$\vdots$	

Fig. 13 Mutual recursion

Expression	Comments
rec $x =_{[n]} \{Head = 0, Tail = x\}$ **in** x	This is a valid answer, representing an infinite (cyclic) list of zeroes.

Fig. 14 Recursive data structure

variable, this variable has to be substituted with its value in order for evaluation to continue. Figure 12 provides two examples differing only by one size indication. In the first case, the expression is a valid answer. In the second case, at the level of compiled code, a block is pre-allocated for *z*, which will eventually represent its value, so we must update it: the value of *y* is copied to this block. At the source language level, this copying enables $\lambda_\circ$ to correctly reflect sharing in the compiled code, and therefore makes it ready for extension with mutable values.

Figure 13 shows an example of mutually recursive functions, assuming that $\lambda_\circ$ has been extended with standard operations on booleans and integers. Finally, one may wonder why we do not perform substitution immediately after evaluation, as usual, but use destruct-time substitution instead. The reason is that it better represents the semantics of the construct we want to define. First, as previously mentioned, sharing is propertly modeled. Second, as shown in Figure 14, it allows to represent recursive data structures such as infinite lists.

2.4.2 Mixin modules

We now consider a more elaborate example, namely an encoding of a simple language of mixin modules, following the approach of [14]. The design of mixin modules in a call-by-value setting raises a number of issues that fall outside the scope of this paper; see [12] for a discussion. Our goal here is to informally explain why $\lambda_\circ$ is an adequate target language

for compiling mixin modules. Thus, we briefly describe a simple language of call-by-value mixin modules, for which we sketch a compilation scheme.

Mixin modules Mixin modules are unevaluated modules with holes. Mixin modules are to ML-style modules what classes are to objects in object-oriented languages. The language provides a `close` operator to instantiate a complete mixin module into a module, thus triggering the evaluation of its components (see below). In order to obtain a complete mixin module, the language provides modularity operators, such as composition and deletion. For instance, one can define the mixin modules `Even` and `Odd` as follows.

```
mixin Even = import
  odd : int -> int
export
  even x = (x = 0) or (odd (x - 1))
end

mixin Odd = import
  even : int -> int
export
  odd x = (x > 0) and (even (x - 1))
end
```

The holes of a mixin module are called its *imports*, and its defined components are its *exports*. The contents of mixin modules are not evaluated until instantiation, as described below. One can *compose* `Even` and `Odd` to obtain

```
mixin Nat1_Open = Even + Odd
```

which is equivalent to

```
mixin Nat1_Open = import
export
  even x = (x = 0) or (odd (x - 1))
  odd x = (x > 0) and (even (x - 1))
end
```

The name `Nat1_Open` refers to the fact that the definitions of this mixin module are still late bound and can be overridden. Then, this mixin module can be *instantiated* into a proper module by

```
module Nat1 = close Nat1_Open
```

which is equivalent to

```
module Nat1 = struct
  let rec even x = (x = 0) or (odd (x - 1))
  and odd x = (x > 0) and (even (x - 1))
end
```

One can then select components from `Nat1`, and write for instance `Nat1.even 56`.

As an example of *overriding*, one can optimize the definition of `even` in `Nat1_Open` by first removing it from `Nat1_Open`, and then composing the result with a mixin module containing the new definition:

```
mixin Nat2_Open = (Nat1_Open - even) +
  import
  export
    even x = ((x mod 2) = 0)
  end
```

which is equivalent to

```
mixin Nat2_Open = import
export
  odd x = (x > 0) and (even (x - 1))
  even x = ((x mod 2) = 0)
end
```

The obtained mixin module can then be instantiated into a plain module, as above. Finally, we extend `Nat1_Open` with a computation using the defined functions:

```
mixin Nat_Test_Open = Nat1_Open +
  import
    even : int -> int
  export
    test = even 56
  end
```

The obtained mixin module is equivalent to

```
mixin Nat_Test_Open = import
export
  even x = (x = 0) or (odd (x - 1))
  odd x = (x > 0) and (even (x - 1))
  test = even 56
end
```

An incorrect encoding in $\lambda_\circ$ A reasonable idea for encoding mixin modules in $\lambda_\circ$ would be to adapt the standard encoding of objects and classes as recursive records [5]. However, this encoding allows to represent mixin modules, but not to instantiate them. Consider for instance `Nat_Test_Open`. It would be translated into a *generator*, that is, a function over records:

$$\begin{array}{l}\texttt{rec } \mathit{Nat\_Test\_Open} =_{[?]} \lambda \mathit{self}.\\ \quad \{\mathit{even} = \lambda x.(x = 0)\ \texttt{or}\ (\mathit{self}.\mathit{odd}\ (x-1))\\ \quad\ \ \mathit{odd} = \lambda x.(x > 0)\ \texttt{and}\ (\mathit{self}.\mathit{even}\ (x-1))\\ \quad\ \ \mathit{test} = \mathit{self}.\mathit{even}\ 56\ \}\\ \texttt{in}\ \ldots\end{array}$$

Then, the instantiation of `Nat_Test_Open` would consist of taking its fixed point, which gives

$$\texttt{rec } \mathit{Nat\_Test} =_{[n]} \mathit{Nat\_Test\_Open}\ \mathit{Nat\_Test}\ \texttt{in}\ \ldots$$

(assuming n to be the correct size), which gives after substitution

$$\begin{array}{l}\texttt{rec } \mathit{Nat\_Test} = (\lambda \mathit{self}.\{\mathit{even} = \lambda x.\ldots.\mathit{self}.\mathit{odd}\ldots\\ \qquad\qquad\qquad\qquad\quad \mathit{odd} = \lambda x.\ldots.\mathit{self}.\mathit{even}\ldots\\ \qquad\qquad\qquad\qquad\quad \mathit{test} = \mathit{self}.\mathit{even}\ 56\ \})\\ \qquad\qquad\qquad\qquad\quad\ \mathit{Nat\_Test}\\ \texttt{in}\ \ldots\end{array}$$

$$\begin{array}{l}
\longrightarrow^{+} \texttt{rec}\ \mathit{self} =_{[?]} \mathit{Nat_Test} \\
\qquad\qquad \mathit{Nat_Test} =_{[n]} \{\, \mathit{even} = \lambda x. \ldots \mathit{self}.\mathit{odd} \ldots \\
\qquad\qquad\qquad\qquad\quad \mathit{odd} = \lambda x. \ldots \mathit{self}.\mathit{even} \ldots \\
\qquad\qquad\qquad\qquad\quad \mathit{test} = \mathit{self}.\mathit{even}\ 56\,\}) \\
\qquad \texttt{in}\ \ldots \\
\\
\longrightarrow \texttt{rec}\ \mathit{self} =_{[?]} \mathit{Nat_Test} \\
\qquad\qquad \mathit{Nat_Test} =_{[n]} \{\, \mathit{even} = \lambda x. \ldots \mathit{self}.\mathit{odd} \ldots \\
\qquad\qquad\qquad\qquad\quad \mathit{odd} = \lambda x. \ldots \mathit{self}.\mathit{even} \ldots \\
\qquad\qquad\qquad\qquad\quad \mathit{test} = \mathit{Nat_Test}.\mathit{even}\ 56\,\}) \\
\quad \texttt{in}\ \ldots
\end{array}$$

whose evaluation is stuck, because *Nat_Test* is not yet evaluated and its definition is already requested. So the recursive record semantics of objects and classes does not directly adapt to mixin modules. The reason is that the components of a mixin module may strongly depend on each other, in the sense of Section 1.2, while the components of a class are essentially methods, which only weakly depend on each other.

Remark 9 (Objects and strong dependencies) In Java, initialization of instance and static fields by arbitrary expressions can lead to strong dependencies between the fields. However, the semantics of field initialization in Java does not guarantee that a fixed point is reached [11, section 8.3.2.3]. Here is an example.

```
static int f() { return x + 1; }
static int x = f() * 2;
```

This code assigns `2` to `x` instead of causing an error as expected.

A correct encoding in $\lambda_{\circ}$ We must find another way to compile mixin modules. In [14], a mixin module is translated into a record of functions, whose fields correspond to the exports of the source mixin module. Each export is abstracted over the other components upon which it depends, and over a dummy argument, useful for suspending the computation in the absence of dependencies. For instance, the mixin module `Even` has only one export `even`, which depends on the import `odd`, so it is represented by

$$\texttt{rec}\ \mathit{Even} = \{\mathit{even} = \lambda \mathit{odd}.\lambda \_.\lambda x.(x = 0)\ \texttt{or}\ (\mathit{odd}\ (x-1))\}$$

where $\_$ denotes an unused variable. Similarly, `Odd` is represented by

$$\texttt{rec}\ \mathit{Odd} = \{\mathit{odd} = \lambda \mathit{even}.\lambda \_.\lambda x.(x > 0)\ \texttt{and}\ (\mathit{even}\ (x-1))\}$$

The translation of composition merely consists of picking the right fields in the arguments. For example, composing `Even` and `Odd` yields

$$\texttt{rec}\ \mathit{Nat1\_Open} = \{\mathit{even} = \mathit{Even}.\mathit{even}, \mathit{odd} = \mathit{Odd}.\mathit{odd}\}$$

The composition can be generated even in a separate compilation setting, where only the types of `Even` and `Odd` are available. Indeed, it only relies on the names exported by the two mixin modules, which are mentioned in their types. Deletion is as easy as composition, since we only have to pick the non deleted fields of the argument.

Instantiation is more difficult, because of strong dependencies and sizes. Consider for example the instantiation of `Nat_Test_Open`. Here, *even* and *odd* must be defined before *test*, which strongly depends on them. Thus, we obtain

$$\begin{array}{ll} \texttt{rec} & \mathit{even} =_{[?]} \mathit{Nat\_Test\_Open.even}\ \mathit{odd}\ \{\}, \\ & \mathit{odd} =_{[n]} \mathit{Nat\_Test\_Open.odd}\ \mathit{even}\ \{\}, \\ & \mathit{test} =_{[?]} \mathit{Nat\_Test\_Open.test}\ \mathit{even}\ \{\} \\ \texttt{in} & \{\mathit{even} = \mathit{even}, \mathit{odd} = \mathit{odd}, \mathit{test} = \mathit{test}\} \end{array}$$

This translation evaluates as expected, provided we can statically guess the correct size n for the *odd* component. For some data representation strategies, this size can be computed from the static type of *odd*, but not always for other strategies; see Section 7 for a discussion.

Another difficulty of the translation outlined here is to determine a correct order in which to evaluate the components of the mixin being closed. The approach proposed in [14] and refined in [16] relies on exploiting dependency information added to the static types of mixin modules. Another approach, outlined in [12, 13], is to embed dependency information in the run-time representation of mixin modules, and determine a correct evaluation order at run-time.

2.4.3 Recursive modules

Another possible application of $\lambda_\circ$ is for compiling recursive modules in extensions of the ML module system [7, 27, 21, 9]. Recursive structures are easily encoded in $\lambda_\circ$. For example, consider the following two mutually recursive structures:

```
module Even = struct
  let even x = (x = 0) or (Odd.odd (x - 1))
end
and Odd = struct
  let odd x = (x > 0) and (Even.even (x - 1))
end
```

Define the syntactic sugar `struct` $\overline{b}$ `end`, where $\overline{b}$ is a list of declarations of the shape $X_1 \triangleright x_1 \diamond_1 e_1, \ldots, X_n \triangleright x_n \diamond_n e_n$, to denote `rec` $x_1 \diamond_1 e_1, \ldots, x_n \diamond_n e_n$ `in` $\{X_1 = x_1, \ldots, X_n = x_n\}$. Using this notation, the example above can be expressed as

$$\begin{array}{l} \texttt{rec}\ \mathit{Even} =_{[?]} \texttt{struct} \\ \quad \mathit{even} \triangleright \mathit{even} =_{[?]} \lambda x.(x = 0)\ \texttt{or}\ (\mathit{Odd.odd}\ (x-1)) \\ \quad \texttt{end}, \\ \quad \mathit{Odd} =_{[n]} \texttt{struct} \\ \quad \mathit{odd} \triangleright \mathit{odd} =_{[?]} \lambda x.(x > 0)\ \texttt{and}\ (\mathit{Even.even}\ (x-1)) \\ \quad \texttt{end} \\ \texttt{in}\ \ldots \end{array}$$

(where n is assumed to be the right size indication). Notice that the function definitions and the first module do not need to have known sizes, since the only forward reference concerns the second module *Odd*.

Beyond recursive structures, it is desirable to encode recursive functor applications, which appear in many practical uses of recursive modules. For instance, consider the following example, taken from the OCaml documentation [20, section 7.9].

```
module A : sig
    type t = Leaf of string | Node of ASet.t
    val compare: t -> t -> int
  end = struct
    type t = Leaf of string | Node of ASet.t
    let compare t1 t2 = ... ASet.compare ...
  end
and ASet : Set.S with type elt = A.t
         = Set.Make(A)
```

After erasing the type components of structures, we encode this example in $\lambda_\circ$ by

$$\begin{array}{l}\texttt{rec}\ A \triangleright A =_{[?]} \texttt{struct}\\ \quad\quad compare \triangleright compare =_{[?]} \ldots ASet.compare \ldots\\ \quad\quad \texttt{end},\\ \quad\quad ASet \triangleright ASet =_{[n]} Set.Make\ A\\ \texttt{in}\ \ldots\end{array}$$

(where n is, again, assumed to be the right size indication). This expression evaluates correctly because *Set.Make* only weakly depends on its argument. The extension of this encoding to a separate compilation setting does not raise the problem of sizes we had for mixin modules: the sizes of ML modules can be guessed from their types. However, the dependency analysis remains difficult, and we are working on this issue.

This section has demonstrated the expressive power of $\lambda_\circ$ by showing encodings of mixin modules and recursive modules, which attests its expressive power. In order to show how to compile it to efficient machine code, we now define a more elementary language called λ_a, into which we then translate $\lambda_\circ$.

3 The target language λ_a

In this section, we define λ_a, a λ-calculus with explicit heap. It was carefully engineered to map directly to an abstract machine with a heap, and to enable efficient compilation to machine code. In particular, the heaps used in the semantics closely correspond to machine-level heaps. (This is apparent in the size requirement for the update operation to work.)

3.1 Syntax

The syntax of the target language λ_a is presented in Figure 15. It includes the λ-calculus with natural numbers and non-recursive `let` binding. Note that a `let` definition $t = E$ computes E, and then either binds the result (if t is a variable) or ignores it (if $t = \_$). The multiple value binding $\texttt{let}\ t_1 = E_1, \ldots, t_n = E_n\ \texttt{in}\ E$ should be understood as $\texttt{let}\ t_1 = E_1\ \texttt{in} \ldots \texttt{let}\ t_n = E_n\ \texttt{in}\ E$. We write ε for the empty binding. Having a multiple `let` binding contributes to make the equational theory of λ_a rich enough for the immediate in-place update scheme to be correct. Additionally, there are constructs for record operations (creation and selection), and constructs for modeling the heap: an allocation operator `alloc`, and an update operator `update`.

The semantics of λ_a uses a notion of heap, which comes in the form of a kind of global `let rec`. A *raw configuration* C is a pair $\texttt{Rec}\,H\ \texttt{in}\,E$ of a *heap* H and an expression E. A

Variable:	x	$\in$	vars	
Name:	X	$\in$	names	
Expression:	$E \in \mathsf{Expr}$	$::=$	n	Natural number
		$\mid$	$x \mid \lambda x.E \mid E\ E$	λ-calculus
		$\mid$	$\mathtt{let}\ B\ \mathtt{in}\ E$	Non-recursive definitions
		$\mid$	$\{R\} \mid E.X$	Record operations
		$\mid$	$\mathtt{alloc} \mid \mathtt{update}$	Heap operations
Record row:	R	$::=$	$\varepsilon \mid (X = V, R)$	
Binding:	B	$::=$	$\varepsilon \mid (t = E, B)$	
	t	$::=$	$x \mid \_$	Variable or wildcard
Value:	$V \in \mathsf{Values}$	$::=$	$x \mid n$	
Stored value:	$S \in \mathsf{SValues}$	$::=$	$\lambda x.E \mid \mathtt{alloc}\ n \mid \{R\}$	
Heap:	$H \in \mathsf{Heaps}$	$::=$	$\varepsilon \mid x = S, H$	
Configuration:	C	$::=$	$\mathtt{Rec}\ H\ \mathtt{in}\ E$	
Evaluation answer:	$A \in \mathsf{Answers}$	$::=$	$\mathtt{Rec}\ H\ \mathtt{in}\ V$	

Fig. 15 Syntax of λ_{a}

$$
\begin{array}{lll@{\qquad}lll}
\mathsf{FV}(n) & = \emptyset & & \mathsf{FV}(\{R\}) & = \mathsf{FV}(R) \\
\mathsf{FV}(x) & = \{x\} & & \mathsf{FV}(E.X) & = \mathsf{FV}(E) \\
\mathsf{FV}(\lambda x.E) & = \mathsf{FV}(E) \setminus \{x\} & & \mathsf{FV}(\mathtt{alloc}) & = \emptyset \\
\mathsf{FV}(E_1\ E_2) & = \mathsf{FV}(E_1) \cup \mathsf{FV}(E_2) & & \mathsf{FV}(\mathtt{update}) & = \emptyset \\
\mathsf{FV}(\mathtt{let}\ B\ \mathtt{in}\ E) & = \mathsf{FV}(B, \_ = E) \setminus \mathsf{dom}(B) & & & \\
\mathsf{FV}(\varepsilon) & = \emptyset & & \mathsf{FV}(t = E, B) & = \mathsf{FV}(E) \cup \mathsf{FV}(B) \cup (\{t\} \cap \mathsf{vars}) \\
\mathsf{FV}(R) & = \bigcup_{X \in \mathsf{dom}(R)} \mathsf{FV}(R(X)) & & \mathsf{FV}(\mathtt{Rec}\ H\ \mathtt{in}\ E) & = (\mathsf{FV}(H) \cup \mathsf{FV}(E)) \setminus \mathsf{dom}(H) \\
\mathsf{FV}(\varepsilon) & = \emptyset & & \mathsf{FV}(x = S, H) & = \{x\} \cup \mathsf{FV}(S) \cup \mathsf{FV}(H)
\end{array}
$$

Fig. 16 Free variables in λ_{a}

heap is list of bindings $x = S$, where the *stored value* $S \in \mathsf{SValues}$ is either a function $\lambda x.E$, or a record $\{R\}$, or an application of the shape $\mathtt{alloc}\ n$ for some natural number n. A *value* V is either a natural number or a variable (but not a stored value). An evaluation *answer* is a raw configuration of the shape $\mathtt{Rec}\ H\ \mathtt{in}\ V$.

Record rows R, (resp. bindings B and heaps H) are required not to define the same name (resp. variable) twice. We use for them the same notations as for $\lambda_\circ$ record rows and bindings for domain, codomain, concatenation, and so on. Observe that the wildcard $\_$ is not a variable, hence is not in the domain of bindings nor in their free variables.

Structural equivalence Free variables are defined in Figure 16. We call *structural equivalence* the smallest equivalence relation including reordering of heap bindings and renaming of bound variables. We call *configurations* structural equivalence classes of raw configurations. We write $=$ for equality of raw configurations and $\equiv$ for equality of configurations. We extend substitutions to expressions and configurations in the standard way. For defining capture-avoiding substitution on expressions, the only non-trivial case is $\mathtt{let}\ B\ \mathtt{in}\ E$: the application of a substitution to an expression of the shape $\mathtt{let}\ t_1 = E_1, \ldots, t_n = E_n\ \mathtt{in}\ E$ proceeds exactly as applying it to $\mathtt{let}\ t_1 = E_1\ \mathtt{in} \ldots \mathtt{let}\ t_n = E_n\ \mathtt{in}\ E$.

Lift context:	$\eta ::= E\ \Box \mid \Box\ V \mid \Box.X$
Nested lift context:	$\varphi ::= \Box \mid E\ \varphi \mid \varphi\ V \mid \varphi.X$
Evaluation context:	$\xi ::= \varphi \mid \mathtt{let}\ t = \varphi, B\ \mathtt{in}\ E$
Allocation context:	$\alpha ::= \Box \mid \alpha\ E \mid E\ \alpha \mid \alpha.X \mid \mathtt{let}\ B_1, t = \alpha, B_2\ \mathtt{in}\ E \mid \mathtt{let}\ B\ \mathtt{in}\ \alpha$

Fig. 17 Evaluation and allocation contexts of λ_{a}

Alpha equivalence:

$$\frac{\alpha_2 \equiv \alpha_2'}{\alpha_1[\alpha_2] \equiv \alpha_1[\alpha_2']} \qquad \frac{E \equiv E'}{E\ \alpha \equiv E'\ \alpha} \qquad \frac{E \equiv E'}{\alpha\ E \equiv \alpha\ E'}$$

$$\frac{B_1 \equiv B_1' \qquad (\mathtt{let}\ B_2\ \mathtt{in}\ E) \equiv (\mathtt{let}\ B_2'\ \mathtt{in}\ E')}{(\mathtt{let}\ B_1, x \diamond \alpha, B_2\ \mathtt{in}\ E) \equiv (\mathtt{let}\ B_1', x \diamond \alpha, B_2'\ \mathtt{in}\ E')} \qquad \frac{B \equiv B'}{\mathtt{let}\ B\ \mathtt{in}\ \alpha \equiv \mathtt{let}\ B'\ \mathtt{in}\ \alpha}$$

$$\frac{E \equiv E'}{(B_1, x = E, B_2) \equiv (B_1, x = E', B_2)}$$

Free variables:

$$\begin{array}{ll}
\mathsf{FV}(\Box) & = \emptyset \\
\mathsf{FV}(\alpha\ E) & = \mathsf{FV}(\alpha) \cup \mathsf{FV}(E) \\
\mathsf{FV}(E\ \alpha) & = \mathsf{FV}(\alpha) \cup \mathsf{FV}(E) \\
\mathsf{FV}(\alpha.X) & = \mathsf{FV}(\alpha) \\
\mathsf{FV}(\mathtt{let}\ B\ \mathtt{in}\ \alpha) & = \mathsf{FV}(B) \cup \mathsf{FV}(\alpha) \\
\mathsf{FV}(\mathtt{let}\ B_1, t = \alpha, B_2\ \mathtt{in}\ E) & = \mathsf{FV}(B_1) \cup \mathsf{FV}(\alpha) \cup \mathsf{FV}(\mathtt{let}\ B_2\ \mathtt{in}\ E) \cup (\{t\} \cap \mathsf{vars})
\end{array}$$

Captured variables:

$$\begin{array}{ll}
\mathsf{Capt}_{\Box}(\Box) & = \emptyset \\
\mathsf{Capt}_{\Box}(\alpha\ E) & = \mathsf{Capt}_{\Box}(\alpha) \\
\mathsf{Capt}_{\Box}(E\ \alpha) & = \mathsf{Capt}_{\Box}(\alpha) \\
\mathsf{Capt}_{\Box}(\alpha.X) & = \mathsf{Capt}_{\Box}(\alpha) \\
\mathsf{Capt}_{\Box}(\mathtt{let}\ B\ \mathtt{in}\ \alpha) & = \mathsf{dom}(B) \\
\mathsf{Capt}_{\Box}(\mathtt{let}\ B_1, t = \alpha, B_2\ \mathtt{in}\ E) & = \mathsf{dom}(B_1) \cup \mathsf{Capt}_{\Box}(\alpha)
\end{array}$$

Fig. 18 Structural equivalence of λ_{a} allocation contexts

Finally, the free variables of a substitution σ (any function from variables to one of the syntactic classes) are defined by

$$\mathsf{FV}(\sigma) = \bigcup_{x \in \mathsf{supp}(\sigma)} \{x\} \cup \mathsf{FV}(\sigma(x)).$$

3.2 Dynamic semantics

The semantics of λ_{a} is defined by a *reduction relation* $\longrightarrow$, which, like that of $\lambda_{\circ}$, is first defined as a relation over raw configurations, then straightforwardly lifted to a relation over configurations.

3.2.1 The reduction relation

The reduction relation is defined in Figures 17, 18, and 19, using the following hypothesis.

Hypothesis 10 (Size in λ_a) We assume given a function Size from stored values to natural numbers such that

- for all n, $\mathsf{Size}(\texttt{alloc}\ n) = n$, and
- for all $\sigma \in \mathsf{vars} \to \mathsf{Values}$ and S, $\mathsf{Size}(\sigma(S)) = \mathsf{Size}(S)$.

The second condition follows the intuition that the size of a stored value is determined by its top constructor, and is therefore invariant under substitutions (which do not change the top constructor, only its arguments). (It is also of technical use in the proof of correctness.)

The reduction rules are defined in Figure 19, using the notions of contexts defined in Figure 17, and the scoping rules and functions of Figure 18.

Rule $\textsc{Beta}_a$ is unusual in that it applies a heap allocated function to an argument V. The function must be a variable x bound in the heap to a value $\lambda y.E$, and the result is $[y \mapsto V](E)$. The reduction can take place in any evaluation context ξ.

Rule $\textsc{Project}_a$ projects a name X out of a heap allocated record $\{R\}$ at variable x, returning $R(X)$.

Rule $\textsc{Update}_a$ copies the contents (the stored value) of a variable to another variable. Both stored values must have exactly the same size and the copied one must not have the shape $\texttt{alloc}\ n$. This condition may seem unnecessary, but it is used to prove that faultiness is preserved by our translation. Recall that $H\langle x = S\rangle$ denotes H where the binding for x is replaced by $x = S$.

As in $\lambda_\circ$, the evaluation of bindings is confined to the top level of configurations. This requires the $\textsc{Lift}_a$ rule, which lifts a binding outside of a lift context η.

By rule $\textsc{Im}_a$, if the first definition of the top-level binding B is itself a binding $\texttt{let}\ B_1\ \texttt{in}\ E_1$, then B_1 is merged with B.

Rule $\textsc{Let}_a$ describes the top-level evaluation of bindings. Let $[t \mapsto V]$ denote $[x \mapsto V]$ if t is a variable x, and the identity substitution otherwise. Once the first definition is evaluated, if t is a variable, then this variable is replaced with the obtained value in the rest of the expression; if $t = \_$, evaluation proceeds directly. When the binding becomes empty, it can be removed with rule $\textsc{EmptyLet}_a$.

By rule $\textsc{WeakGC}_a$, when a heap binding is not used by any other binding than itself, and not used by the expression either, it can be removed. This is formalized by requiring that the corresponding variable x be outside the set of free variables $\mathsf{FV}(H_{\setminus\{x\}}) \cup \mathsf{FV}(E)$ of other heap bindings and of the main expression. This simple rule is here to model the garbage collection step mentioned in the explanation of Figure 2: it allows garbage-collecting the blocks obtained by evaluation of the recursively-defined expressions once they have been copied to the pre-allocated blocks. A general garbage collection rule could detect more kinds of dead data structures, in particular mutually dependent, otherwise unused data structures. This additional power is not needed in this paper, so we do not have a general garbage collection rule.

Finally, rule $\textsc{Alloc}_a$ is one of the key points of λ_a, by which a configuration of the shape $\texttt{Rec}\,H\ \texttt{in}\ \alpha[S]$ evaluates to the configuration $\texttt{Rec}\,x = S, H\ \texttt{in}\ \alpha[x]$, where x is a fresh variable. In particular, if S is $\texttt{alloc}\ n$, the evaluation allocates a dummy block of size n on the heap. This reduction can happen in any *allocation context* α. Allocation contexts cover all contexts of λ_a, except under λ-abstractions. The idea is that a value can be allocated in advance in the heap. For instance, given a configuration $\texttt{Rec}\,H\ \texttt{in}\ \texttt{let}\ B\ \texttt{in}\ S$, it is possible to allocate S before computing the binding, provided S does not use the variables defined in B. The side condition $\mathsf{FV}(S)\ \#\ \mathsf{Capt}_\Box(\alpha)$ ensures this, where $\mathsf{Capt}_\Box(\alpha)$ denotes the set of binders located above the context hole in α, here $\mathsf{dom}(B)$ (see Figure 18).

$$\frac{H(x) = \lambda y.E}{\mathtt{Rec}\,H\ \mathtt{in}\ \xi[x\,V] \longrightarrow \mathtt{Rec}\,H\ \mathtt{in}\ \xi[[y \mapsto V](E)]}\ (\textsc{Beta}_{\mathsf{a}})$$

$$\frac{H(x) = \{R\}}{\mathtt{Rec}\,H\ \mathtt{in}\ \xi[x.X] \longrightarrow \mathtt{Rec}\,H\ \mathtt{in}\ \xi[R(X)]}\ (\textsc{Project}_{\mathsf{a}})$$

$$\frac{H(y) \notin \{\mathtt{alloc}\ n \mid n \in \mathbb{N}\} \qquad \mathsf{Size}(H(y)) = \mathsf{Size}(H(x))}{\mathtt{Rec}\,H\ \mathtt{in}\ \xi[\mathtt{update}\ x\,y] \longrightarrow \mathtt{Rec}\,H\langle x = H(y)\rangle\ \mathtt{in}\ \xi[\{\}]}\ (\textsc{Update}_{\mathsf{a}})$$

$$\frac{\mathsf{dom}(B)\,\#\,\mathsf{FV}(\eta)}{\mathtt{Rec}\,H\ \mathtt{in}\ \xi[\eta[\mathtt{let}\ B\ \mathtt{in}\ E]] \longrightarrow \mathtt{Rec}\,H\ \mathtt{in}\ \xi[\mathtt{let}\ B\ \mathtt{in}\ \eta[E]]}\ (\textsc{Lift}_{\mathsf{a}})$$

$$\frac{\mathsf{dom}(B_1)\,\#\,\{t\} \cup \mathsf{FV}(B_2) \cup \mathsf{FV}(E_2)}{\begin{array}{l}\mathtt{Rec}\,H\ \mathtt{in}\ \mathtt{let}\ t = (\mathtt{let}\ B_1\ \mathtt{in}\ E_1), B_2\ \mathtt{in}\ E_2 \\ \longrightarrow \mathtt{Rec}\,H\ \mathtt{in}\ \mathtt{let}\ B_1, t = E_1, B_2\ \mathtt{in}\ E_2\end{array}}\ (\textsc{IM}_{\mathsf{a}}) \qquad \begin{array}{l}\mathtt{Rec}\,H\ \mathtt{in}\ \mathtt{let}\ t = V, B\ \mathtt{in}\ E \\ \longrightarrow \mathtt{Rec}\,H\ \mathtt{in}\ [t \mapsto V](\mathtt{let}\ B\ \mathtt{in}\ E)\end{array}\ (\textsc{Let}_{\mathsf{a}})$$

$$\mathtt{Rec}\,H\ \mathtt{in}\ \mathtt{let}\ \varepsilon\ \mathtt{in}\ E \longrightarrow \mathtt{Rec}\,H\ \mathtt{in}\ E\ \ (\textsc{EmptyLet}_{\mathsf{a}}) \qquad \frac{x \notin (\mathsf{FV}(H_{\setminus\{x\}}) \cup \mathsf{FV}(E))}{\mathtt{Rec}\,H\ \mathtt{in}\ E \longrightarrow \mathtt{Rec}\,H_{\setminus\{x\}}\ \mathtt{in}\ E}\ (\textsc{WeakGC}_{\mathsf{a}})$$

$$\frac{x \notin \mathsf{FV}(S) \cup \mathsf{FV}(\alpha) \cup \mathsf{FV}(H) \qquad \mathsf{FV}(S)\,\#\,\mathsf{Capt}_{\square}(\alpha)}{\mathtt{Rec}\,H\ \mathtt{in}\ \alpha[S] \longrightarrow \mathtt{Rec}\,x = S, H\ \mathtt{in}\ \alpha[x]}\ (\textsc{Alloc}_{\mathsf{a}})$$

Fig. 19 Dynamic semantics of λ_{a}

Remark 11 (Non-determinism and evaluation order) Unlike in $\lambda_{\circ}$, the reduction of λ_{a} is not deterministic because of rules $\textsc{WeakGC}_{\mathsf{a}}$ and $\textsc{Alloc}_{\mathsf{a}}$. Nevertheless, λ_{a} remains close to an abstract machine, which would simply implement a particular reduction strategy. Furthermore, this non-determinism makes the equational theory of λ_{a} rich enough for the correctness proof of Section 5.

Although λ_{a} is not deterministic, function applications are evaluated from right-to-left, because of the lift contexts $\square\ V$ and $E\ \square$. This makes the presentation more concise, since it avoids lift contexts of the shape $\mathtt{alloc}\ \square$, $\mathtt{update}\ \square$, and $\mathtt{update}\ x\ \square$, and explains why $\lambda_{\circ}$ also evaluates its arguments from right to left. The results of the paper can be adapted to a left-to-right evaluation setting with some additional work.

3.2.2 Confluence and errors

Since reduction in λ_{a} is not deterministic, it is important to make sure that it is confluent. In fact, we show that the reduction relation is strongly commuting, which implies that it is confluent by Hindley's lemma.

Lemma 12 (The reduction rules are strongly commuting) *For all reduction rules R_1, R_2, and configurations C, C_1, C_2, if $C \xrightarrow{R_1} C_1$ and $C \xrightarrow{R_2} C_2$, then there exists C' such that $C_1 \xrightarrow{R_2} C'$ and $C_2 \xrightarrow{R_1} C'$.*

Proof By case analysis on the possible pairs of reductions. The reduction relation without rules $\textsc{WeakGC}_{\mathsf{a}}$ and $\textsc{Alloc}_{\mathsf{a}}$ is deterministic, so we only have to examine the pairs involving at least one of these rules. □

Expression	Comments
$\texttt{Rec}\,\varepsilon\,\texttt{in}$ $(\lambda x.(x.X.Y))$ $\quad(\texttt{let}\,y = \{Y = 0\}\,\texttt{in}\,\{X = y\})$	Before applying rule BETA_a, we must reduce the function and the argument to values. For this, we apply (several possible orders) rules ALLOC_a (three times), LET_a and EMPTYLET_a.
$\downarrow_+$	
$\texttt{Rec}\left\{\begin{array}{l} x_1 = \{Y = 0\}, \\ x_2 = \{X = x_1\}, \\ x_3 = \lambda x.(x.X.Y) \end{array}\right\}\texttt{in}$ $x_3\ x_2$	We then apply rule BETA_a (the heap H remains unchanged).
$\downarrow$	
$\texttt{Rec}\,H\,\texttt{in}\,x_2.X.Y$	We finally apply rule PROJECT_a twice.
$\downarrow_+$	
$\texttt{Rec}\,H\,\texttt{in}\,0$	

Fig. 20 An example of reduction in λ_a

A configuration is said to be *faulty* if it reduces to a configuration in normal form that is not in Answers. For a better understanding of the semantics, we now characterize the set of faulty configurations.

Proposition 13 (Faulty λ_a configurations) *A configuration is faulty iff it reduces to a configuration C in normal form such that:*

- $C \equiv \texttt{Rec}\,H\,\texttt{in}\,\xi[x\ V]$, *with either*
  - $x \notin \mathsf{dom}(H)$, *or*
  - $H(x)$ *is not a function,*
- $C \equiv \texttt{Rec}\,H\,\texttt{in}\,\xi[n\ V]$,
- *or* $C \equiv \texttt{Rec}\,H\,\texttt{in}\,\xi[x.X]$, *with either*
  - $x \notin \mathsf{dom}(H)$, *or*
  - $H(x)$ *is not a record with field* X,
- *or* $C \equiv \texttt{Rec}\,H\,\texttt{in}\,\xi[n.X]$,
- *or* $C \equiv \texttt{Rec}\,H\,\texttt{in}\,\xi[\texttt{alloc}]$ *and* $\xi \neq \alpha'[\Box\ n]$, *for all* α', n,
- *or* $C \equiv \texttt{Rec}\,H\,\texttt{in}\,\xi[\texttt{update}\ x\ y]$, *with either*
  - x *or* y *not in* $\mathsf{dom}(H)$, *or*
  - x *and* y *have different sizes, i.e.,* $\mathsf{Size}(H(x)) \neq \mathsf{Size}(H(y))$, *or*
  - $H(y)$ *of the shape* $\texttt{alloc}\ n$,
- *or* $C \equiv \texttt{Rec}\,H\,\texttt{in}\,\xi[\texttt{update}]$ *and* $\xi \neq \xi'[\Box\ x\ y]$, *for all* ξ', x, y.

3.3 Examples

Figure 20 exemplifies the evaluation of a function application in λ_a. The function selects the Y field of the X field of its argument. However, in λ_a, neither the function nor the argument are considered values. The evaluation of the argument $(\texttt{let}\,y = \{Y = 0\}\,\texttt{in}\,\{X = y\})$ involves two heap allocations: first, $x_1 = \{Y = 0\}$ is allocated; then, we apply rules LET_a and EMPTYLET_a; finally, we allocate $x_2 = \{X = x_1\}$. The evaluation of the function $\lambda x.(x.X.Y)$ involves one heap allocation $x_3 = \lambda x.(x.X.Y)$. The executed expression is then $x_3\ x_2$, which reduces in one step to $x_2.X.Y$, and then in two steps to 0.

Expression	Comments
$\texttt{Rec}\,\varepsilon\,\texttt{in}$ $\texttt{let}\ \ odd = \texttt{alloc}\ n,$ $\quad even = \lambda x.\ (x = 0)\ \texttt{or}$ $\quad\quad\quad (odd\ (x-1)),$ $\quad \_ = \texttt{update}\ odd$ $\quad\quad \lambda x.\ (x > 0)\ \texttt{and}$ $\quad\quad\quad (even\ (x-1)),$ $\texttt{in}\ even\ 56$	We pre-allocate a block for *odd*, evaluate *even* (which points to the dummy block), then evaluate the definition of *odd* and update the dummy block with it.
$\downarrow_+$	
$\texttt{Rec}\left\{\begin{array}{l} x_1 = \texttt{alloc}\ n, \\ x_2 = \lambda x.\ (x = 0)\ \texttt{or} \\ \quad\quad (x_1\ (x-1)), \\ x_3 = \lambda x.\ (x > 0)\ \texttt{and} \\ \quad\quad (x_2\ (x-1)) \end{array}\right\}\texttt{in}$ $\texttt{let}\ \_ = \texttt{update}\ x_1\ x_3$ $\texttt{in}\ x_2\ 56$	First, the two evaluated heap blocks defining *odd* and *even* are allocated, yielding x_1 and x_2, respectively. Then, the second argument of `update` is allocated, yielding x_3.
$\downarrow_+$	
$\texttt{Rec}\left\{\begin{array}{l} x_1 = \lambda x.\ (x > 0)\ \texttt{and} \\ \quad\quad (x_2\ (x-1)), \\ x_2 = \lambda x.\ (x = 0)\ \texttt{or} \\ \quad\quad (x_1\ (x-1)), \end{array}\right\}\texttt{in}$ $x_2\ 56$	Now x_1 is updated with x_3, which can then be garbage-collected, and the evaluation can proceed with the two expected mutually recursive functions.

Fig. 21 Mutually recursive functions in λ_{a} (compare with Figure 13)

Translation of expressions:	$[\![x]\!]$	$\equiv x$
	$[\![\lambda x.e]\!]$	$\equiv \lambda x.[\![e]\!]$
	$[\![e_1\ e_2]\!]$	$\equiv [\![e_1]\!]\ [\![e_2]\!]$
	$[\![\{r\}]\!]$	$\equiv \{r\}$
	$[\![e.X]\!]$	$\equiv [\![e]\!].X$
	$[\![\texttt{rec}\ b\ \texttt{in}\ e]\!]$	$\equiv \texttt{let}\ \mathsf{Dummy}(b), \mathsf{Update}(b)\ \texttt{in}\ [\![e]\!]$
Pre-allocation of bindings:	$\mathsf{Dummy}(\varepsilon)$	$\equiv \varepsilon$
	$\mathsf{Dummy}(x =_{[n]} e, b)$	$\equiv (x = \texttt{alloc}\ n, \mathsf{Dummy}(b))$
	$\mathsf{Dummy}(x =_{[?]} e, b)$	$\equiv \mathsf{Dummy}(b)$
Computation of bindings:	$\mathsf{Update}(\varepsilon)$	$\equiv \varepsilon$
	$\mathsf{Update}(x =_{[n]} e, b)$	$\equiv (\_ = (\texttt{update}\ x\ [\![e]\!]), \mathsf{Update}(b))$
	$\mathsf{Update}(x =_{[?]} e, b)$	$\equiv (x = [\![e]\!], \mathsf{Update}(b))$

Fig. 22 Standard translation from $\lambda_{\circ}$ to λ_{a}

Figure 21 shows the evaluation of a mutually recursive function definition. It is the λ_{a} analogue of the example shown earlier in Figure 13.

4 Compilation

4.1 The standard translation

We now define a translation from $\lambda_{\circ}$ to λ_{a} that straightforwardly implements the in-place update trick. This translation, called the *standard* translation, is defined in Figure 22.

The translation is straightforward for variables, functions, applications, and record operations. The translation of a binding b is the concatenation of two λ_{a} bindings. The first binding $\mathsf{Dummy}(b)$ is called the *pre-allocation* binding, and gives instructions to allocate dummy blocks on the heap for definitions of known sizes. The second binding $\mathsf{Update}(b)$ is called the *update* binding. It evaluates the definitions and either updates the previously pre-allocated dummy blocks for definitions of known sizes, or simply binds the result for definitions of unknown sizes.

Example 14 The standard translation of the first expression of Figure 13 is (part of) the first configuration of Figure 21:

$$\texttt{rec}\left(\begin{array}{l} \mathit{even} =_{[?]} \lambda x.\,(x = 0)\ \texttt{or} \\ \qquad\qquad (\mathit{odd}\ (x-1)), \\ \mathit{odd} =_{[n]} \lambda x.\,(x > 0)\ \texttt{and} \\ \qquad\qquad (\mathit{even}\ (x-1)) \end{array}\right) \texttt{in}\ \mathit{even}\ 56$$

is translated to

$$\texttt{let}\left(\begin{array}{l} \mathit{odd} = \texttt{alloc}\ n, \\ \mathit{even} = \lambda x.\,(x = 0)\ \texttt{or} \\ \qquad\qquad (\mathit{odd}\ (x-1)), \\ \_ = \texttt{update}\,\mathit{odd} \\ \quad (\lambda x.\,(x > 0)\ \texttt{and} \\ \qquad\qquad (\mathit{even}\ (x-1))), \end{array}\right) \texttt{in}\ \mathit{even}\ 56$$

Remark 15 (Restriction on forward references in $\lambda_{\circ}$) The standard translation crucially relies on the fact that $\lambda_{\circ}$ forbids forward references to definitions of unknown sizes: such forward references, after translation, would produce references to unbound variables. For example, consider the illegal binding $x =_{[?]} y, y =_{[?]} e$. Its pre-allocation pass is empty, and it is translated as $x = y, y = [\![e]\!]$, where y is unbound. (Recall that λ_{a} bindings do not have a recursive scope.)

For any reduction rule R, write $\xrightarrow{R}$ for the set of pairs of expressions or configurations that are instances of R.

Proposition 16 *For all $v \in \mathsf{values} \setminus \mathsf{vars}$, there exist H, x such that*

$$\texttt{Rec}\,\varepsilon\,\texttt{in}\,[\![v]\!] \xrightarrow{\textsc{Alloc}_{\mathsf{a}}} \texttt{Rec}\,H\,\texttt{in}\,x$$

Proof By case analysis on v. □

From now on, we assume that the notions of size in $\lambda_{\circ}$ and λ_{a} are coherent, in the following sense.

Hypothesis 17 (Size) For all H, x, and $v \in \mathsf{values} \setminus \mathsf{vars}$, if $\texttt{Rec}\,\varepsilon\,\texttt{in}\,[\![v]\!] \longrightarrow^* \texttt{Rec}\,H\,\texttt{in}\,x$, then $\mathsf{size}(v) = \mathsf{Size}(H(x))$.

Our main result is:

Theorem 18 (Correctness) *For all e, if e reduces to an answer, loops, or is faulty in $\lambda_{\circ}$ then so does $[\![e]\!]$ in λ_{a}.*

The rest of the paper is devoted to proving this theorem. This raises several difficulties, which we explain before actually delving into the proof.

4.2 Overview of difficulties

A natural approach to proving the correctness of our translation is to use a simulation argument: if $e \longrightarrow e'$ in $\lambda_\circ$, then $[\![e]\!] \longrightarrow^+ [\![e']\!]$; moreover, if e is an answer, $[\![e]\!]$ should be an answer as well. However, both properties fail, for reasons illustrated in the following examples.

Example 19 (Administrative reductions) Consider $e \equiv \lambda x.x$. Its translation is $E \equiv \lambda x.x$, which is not an answer. An allocation has to be performed in order to reduce it to the answer $\mathtt{Rec}\, y = \lambda x.x\, \mathtt{in}\, y$. In general, the translation of a $\lambda_\circ$ value reduces in a finite number of $\textsc{Alloc}_\mathsf{a}$ steps to a λ_a answer.

Example 20 (More administrative reductions) Consider $e_1 \equiv \mathtt{rec}\, y =_{[n]} \lambda x.x\, \mathtt{in}\, e_2$, where $n = \mathsf{size}(\lambda x.x)$. If $e_2 \rightsquigarrow e_2'$, then e_1 reduces to $e_1' \equiv \mathtt{rec}\, y =_{[n]} \lambda x.x\, \mathtt{in}\, e_2'$ in $\lambda_\circ$. However, the translations of e_1 and e_1' are

$$[\![e_1]\!] \equiv \mathtt{let}\, y = \mathtt{alloc}\ n, \_ = \mathtt{update}\ y\ (\lambda x.x)\ \mathtt{in}\, [\![e_2]\!]$$
$$[\![e_1']\!] \equiv \mathtt{let}\, y = \mathtt{alloc}\ n, \_ = \mathtt{update}\ y\ (\lambda x.x)\ \mathtt{in}\, [\![e_2']\!]$$

and $[\![e_1]\!]$ does not reduce to $[\![e_1']\!]$ in λ_a: it is generally not possible to reduce $[\![e_2]\!]$ until the enclosing `let` has been fully evaluated. So, if evaluation in $\lambda_\circ$ occurs under a size-respecting binding, then in the compiled code the evaluation of this binding requires a finite number of $\textsc{Alloc}_\mathsf{a}$, $\textsc{Update}_\mathsf{a}$, $\textsc{Let}_\mathsf{a}$, $\textsc{EmptyLet}_\mathsf{a}$, and $\textsc{WeakGC}_\mathsf{a}$ steps, which are exactly the same in $[\![e_1]\!]$ and $[\![e_1']\!]$.

In order to deal with these administrative reductions, we will introduce another translation function, called the *top-level translation,* which performs them on the fly. This is directly inspired by Plotkin's *colon translation* [26]. However, there are other complications that we now illustrate, writing $\lfloor e \rfloor^{\textsc{top}}$ for the top-level translation.

Example 21 (Granularity) Consider $e \equiv (\mathtt{rec}\, x =_{[?]} \lambda y.y\, \mathtt{in}\, x\ z)$. It reduces by rule $\textsc{Subst}_\circ$ to $e' \equiv (\mathtt{rec}\, x =_{[?]} \lambda y.y\, \mathtt{in}\, (\lambda y.y)\ z)$, and then by rule $\textsc{Beta}_\circ$ to $e'' \equiv (\mathtt{rec}\, x =_{[?]} \lambda y.y\, \mathtt{in}\, \mathtt{rec}\, y =_{[?]} z\, \mathtt{in}\, y)$. Remark that rule $\textsc{Subst}_\circ$ duplicates $\lambda y.y$, which is not innocent w.r.t. the translation: $\lfloor e \rfloor^{\textsc{top}}$ does not reduce to $\lfloor e' \rfloor^{\textsc{top}}$. Thus, rule $\textsc{Subst}_\circ$ alone is not simulated. In the compiled code, abstracting over the administrative reductions, there is no substitution: rule $\textsc{Beta}_\mathsf{a}$ is applied directly, fetching the value of x from the heap. Initially, we have something like $\mathtt{Rec}\, H\, \mathtt{in}\, x'\ z$, where $H(x') \equiv \lambda y.y$, which reduces in one step to $\mathtt{Rec}\, H\, \mathtt{in}\, z$.

Example 22 (Beta and the top-level binding) From Example 21, one could expect that although rule $\textsc{Subst}_\circ$ is not exactly simulated, the combination of rules $\textsc{Subst}_\circ$ and $\textsc{Beta}_\circ$ is. This is not the case, because rule $\textsc{Beta}_\circ$ leaves a fully evaluated binding right where the subreduction happened, which is not necessarily at top-level. Consider again Example 21: we have seen that $\lfloor e \rfloor^{\textsc{top}}$ is a configuration of the shape $\mathtt{Rec}\, H\, \mathtt{in}\, x'\ z$, where $H(x') \equiv \lambda y.y$, which reduces in one step to $C \equiv \mathtt{Rec}\, H\, \mathtt{in}\, z$. However, after applying $\textsc{Subst}_\circ$ and $\textsc{Beta}_\circ$ to e, we obtain $e'' \equiv (\mathtt{rec}\, x =_{[?]} \lambda y.y\, \mathtt{in}\, \mathtt{rec}\, y =_{[?]} z\, \mathtt{in}\, y)$, where the inner `rec` is not at top level. Hence, $\lfloor e'' \rfloor^{\textsc{top}}$ is $\mathtt{Rec}\, H\, \mathtt{in}\, \mathtt{let}\, y = z\, \mathtt{in}\, y$, which is different from C. Nevertheless, applying $\textsc{EM}_\circ$ to e'', we obtain $e''' \equiv \mathtt{rec}\, x =_{[?]} \lambda y.y, y =_{[?]} z\, \mathtt{in}\, y$, whose top-level translation is exactly C. More generally, it turns out that enough reduction sequences consisting of applications of $\textsc{Subst}_\circ$, $\textsc{Beta}_\circ$, and a combination of $\textsc{Lift}_\circ$, $\textsc{IM}_\circ$, and $\textsc{EM}_\circ$ are simulated by $\lfloor \cdot \rfloor^{\textsc{top}}$.

Example 23 (Stuttering reductions) In some cases, we have $e \longrightarrow e'$, but $\lfloor e \rfloor^{\text{TOP}} \equiv \lfloor e' \rfloor^{\text{TOP}}$. For instance, consider e of the shape $e \equiv \texttt{rec}\ b_{\mathsf{v}}\ \texttt{in}\ \texttt{rec}\ x =_{[?]} (\texttt{rec}\ b\ \texttt{in}\ e_1)\ \texttt{in}\ e_2$. By rule $\text{EM}_\circ$, e reduces to $e' \equiv \texttt{rec}\ b_{\mathsf{v}}, x =_{[?]} (\texttt{rec}\ b\ \texttt{in}\ e_1)\ \texttt{in}\ e_2$. In fact, in both cases, $\lfloor \cdot \rfloor^{\text{TOP}}$ translates b_{v} on the fly, so that $\lfloor e \rfloor^{\text{TOP}} \equiv \lfloor e' \rfloor^{\text{TOP}}$. Thus, the preservation of non-termination is not trivial.

Example 24 (Lifting and allocation) Let $b \equiv (y =_{[?]} (\lambda x_2.x_2)\ z)$ and consider $e \equiv (\lambda x_1.x_1)\ (\texttt{rec}\ b\ \texttt{in}\ y)$, which reduces by rule $\text{LIFT}_\circ$ to $e' \equiv \texttt{rec}\ b\ \texttt{in}\ (\lambda x_1.x_1)\ y$. Anticipating again the definition of $\lfloor \cdot \rfloor^{\text{TOP}}$ below, in e, $\lambda x_1.x_1$ appears at top-level, and is therefore allocated on the fly, but not $\lambda x_2.x_2$, so we obtain

$$C \equiv \lfloor e \rfloor^{\text{TOP}} \equiv \texttt{Rec}\, x = \lambda x_1.x_1\ \texttt{in}\ x\ (\texttt{let}\ y = (\lambda x_2.x_2)\ z\ \texttt{in}\ y).$$

On the other hand, in e', $\lambda x_2.x_2$ appears at top-level, but not $\lambda x_1.x_1$, which lies below a not fully evaluated binding, so we have

$$C' \equiv \lfloor e' \rfloor^{\text{TOP}} \equiv \texttt{Rec}\, x' = \lambda x_2.x_2\ \texttt{in}\ \texttt{let}\ y = x'\ z\ \texttt{in}\ (\lambda x_1.x_1)\ y.$$

Thus, some $\text{ALLOC}_{\mathsf{a}}$ reductions performed in $\lfloor e \rfloor^{\text{TOP}}$ are not performed in $\lfloor e' \rfloor^{\text{TOP}}$. Here, C reduces by LIFT_{a} and $\text{ALLOC}_{\mathsf{a}}$ to $\texttt{Rec}\, x = \lambda x_1.x_1, x' = \lambda x_2.x_2\ \texttt{in}\ \texttt{let}\ y = x'\ z\ \texttt{in}\ x\ y$, which can be reached from C' by $\text{ALLOC}_{\mathsf{a}}$.

4.3 Overview of the correctness proof

Here is how we deal with these difficulties. First, Example 24 shows that no small-step simulation holds, so we adopt a less accurate notion of observation, namely evaluation answers and non-termination:

- if e reduces to an answer a, then its translation reduces to some λ_{a} answer related to a;
- if e reduces infinitely, then so does its translation.

In order to prove this result, we consider some of the reduction rules of $\lambda_\circ$ and λ_{a} as structural, i.e., not counting as proper reduction steps. This eliminates almost all the difficulties and preserves our notion of observation. The only remaining difficulty is that of Example 21, which we cannot solve in the same way. Indeed, we neither want $\text{SUBST}_\circ$ nor $\text{BETA}_\circ$ and $\text{PROJECT}_\circ$ to be considered structural, as we now explain. First, deeming $\text{BETA}_\circ$ structural would prevent us from proving that non termination is preserved (and doing so for $\text{PROJECT}_\circ$ is thus only a partial, unsatisfactory solution). Furthermore, the equational theory of λ_{a} is not rich enough to equate $\lfloor e \rfloor^{\text{TOP}}$ and $\lfloor e' \rfloor^{\text{TOP}}$ when $e \xrightarrow{\text{SUBST}_\circ} e'$. Indeed, this would involve a currently forbidden duplication ("unsharing") of a stored value. It seems possible to extend λ_{a} in a meaningful way, so as to support unsharing of stored values in some cases. It also seems possible to modify the semantics of $\lambda_\circ$ to avoid duplication before rules $\text{BETA}_\circ$ and $\text{PROJECT}_\circ$. However, the spirit of this article is to keep the source and target languages as standard as possible, which rules out these solutions. Our solution is to consider bigger steps as atomic in $\lambda_\circ$: we consider atomic a sequence of applications of $\text{SUBST}_\circ$, followed by an application of $\text{BETA}_\circ$ or $\text{PROJECT}_\circ$, followed by possible applications of $\text{LIFT}_\circ$, and terminated by a possible application of $\text{IM}_\circ$ or $\text{EM}_\circ$ (to lift a possible binding created by application of $\text{BETA}_\circ$ and merge it with the top-level binding).

We now outline the main steps of the proof, detailed in Section 5.

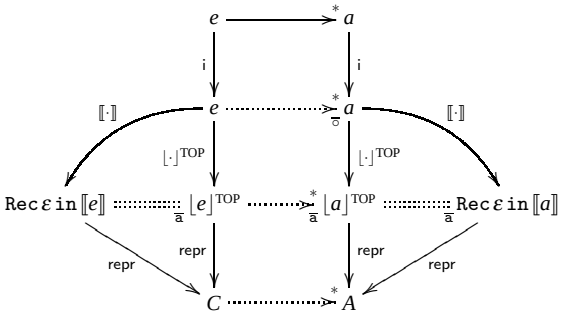


Fig. 23 Summary of the proof (for observation of evaluation answers)

The top-level translation We start by defining the top-level translation $\lfloor\cdot\rfloor^{\text{TOP}}$, based on an enriched notion of context in λ_{a}, which lends itself better than the standard translation to a simulation argument.

Quotient of λ_{a} Then, we consider $\overline{\lambda}_{\mathsf{a}}$, defined as λ_{a} modulo rules UPDATE$_{\mathsf{a}}$, LET$_{\mathsf{a}}$, EMPTYLET$_{\mathsf{a}}$, WEAKGC$_{\mathsf{a}}$, and ALLOC$_{\mathsf{a}}$. These rules are strongly normalizing, and we define a faithful translation from $\overline{\lambda}_{\mathsf{a}}$ to λ_{a}, by taking normal forms as representatives of equivalence classes. Furthermore, the translations $[\![\cdot]\!]$ and $\lfloor\cdot\rfloor^{\text{TOP}}$ are well-defined from $\lambda_{\circ}$ to $\overline{\lambda}_{\mathsf{a}}$, by composition with the canonical injection from λ_{a} to $\overline{\lambda}_{\mathsf{a}}$. Define $=_{\overline{\mathsf{a}}}$ as the equality in $\overline{\lambda}_{\mathsf{a}}$, i.e., the equality of equivalence classes. We then show two crucial properties gained by taking the quotient. First, we abstract over the administrative reductions: for any e, $[\![e]\!] =_{\overline{\mathsf{a}}} \lfloor e\rfloor^{\text{TOP}}$. Second, we make the translation compositional: for all e, $\mathbb{E}$, $\lfloor\mathbb{E}[e]\rfloor^{\text{TOP}} =_{\overline{\mathsf{a}}} \lfloor\mathbb{E}\rfloor^{\text{TOP}}[[\![e]\!]]$. This addresses the problems illustrated by Examples 19, 20, and 24.

Quotient of $\lambda_{\circ}$ Then, we modify the notion of evaluation of $\lambda_{\circ}$ by merging rule SUBST$_{\circ}$ with the immediately following rules. We obtain a language where, instead of first copying the value of a variable and then reducing, we perform the reduction exactly as in λ_{a} by fetching the value from the heap, applying the appropriate rule, and, in the case of beta reduction, merging the obtained binding with the top-level one (all this in one step). This language correctly simulates $\lambda_{\circ}$, since it reaches the same values, diverges on the same expressions, and goes wrong on the same expressions. This addresses the problems described in Examples 21 and 22. Then, we quotient the obtained language by rule EM$_{\circ}$. This gives a language called $\overline{\lambda}_{\circ}$ which also simulates $\lambda_{\circ}$, eliminating the issue raised by Example 23.

Correctness Finally, $\lfloor\cdot\rfloor^{\text{TOP}}$, as a function from $\overline{\lambda}_{\circ}$ to $\overline{\lambda}_{\mathsf{a}}$, yields a simulation. Writing $\longrightarrow_{\overline{\circ}}$ for reduction in $\overline{\lambda}_{\circ}$, $\longrightarrow_{\overline{\mathsf{a}}}$ for reduction in $\overline{\lambda}_{\mathsf{a}}$, i for the injection from $\lambda_{\circ}$ into $\overline{\lambda}_{\circ}$, and $\mathsf{repr}(C)$ for the normal form of C modulo rules UPDATE$_{\mathsf{a}}$, LET$_{\mathsf{a}}$, EMPTYLET$_{\mathsf{a}}$, WEAKGC$_{\mathsf{a}}$, and ALLOC$_{\mathsf{a}}$, the proof may be summarized as in Figure 23 (for observation of evaluation answers), where the dotted arrows and equal signs corresponds to intermediate results.

5 Correctness

5.1 The top-level translation

5.1.1 Overview

We first define the top-level translation from $\lambda_\circ$ to $\lambda_\mathtt{a}$. We start with a simple example.

Example 25 The top-level translation of the first expression of Figure 13 is the last configuration of Figure 21:

$$\mathtt{rec}\begin{pmatrix} \mathit{even} =_{[?]} \lambda x.\,(x = 0)\ \mathtt{or} \\ \qquad\qquad (\mathit{odd}\ (x-1)), \\ \mathit{odd} =_{[n]} \lambda x.\,(x > 0)\ \mathtt{and} \\ \qquad\qquad (\mathit{even}\ (x-1)) \end{pmatrix} \mathtt{in}\ \mathit{even}\ 56$$

gives

$$\mathtt{Rec}\begin{pmatrix} x_1 = \lambda x.\,(x > 0)\ \mathtt{and} \\ \qquad\qquad (x_2\ (x-1)), \\ x_2 = \lambda x.\,(x = 0)\ \mathtt{or} \\ \qquad\qquad (x_1\ (x-1)), \end{pmatrix} \mathtt{in}\ x_2\ 56$$

Roughly, we consider three levels in the translated expression e.

Top-level The first level consists of the (possibly empty) fully evaluated part b_v of the top-level binding of e, if any. At this level, $\lfloor\cdot\rfloor^{\mathrm{TOP}}$ performs all administrative reductions, as previewed in Examples 19 and 20. Hence, the top-level translation maps b_v to a pair of a heap and a substitution, representing the heap after evaluation of the standard translation of b_v, plus the successive substitutions produced by this evaluation. For instance, if $b_\mathsf{v} \equiv (y =_{[n]} \lambda x.\ldots y\ldots)$, then its standard translation is $y = \mathtt{alloc}\ n, \_ = \mathtt{update}\ y\ (\lambda x.\ldots y\ldots)$. Its top-level translation gives directly what we would obtain after performing the admninistrative reductions, i.e., the heap $y' = \lambda x.\ldots y'\ldots$ and the substitution $[y \mapsto y']$.

Allocating After b_v, we expect $\lfloor\cdot\rfloor^{\mathrm{TOP}}$ to map answers to answers. Thus, we also want administrative reductions to be performed on the fly. The difference with the previous level is that (although we could do it) we do not perform administrative reductions on `rec`'s. Indeed, it is not necessary, and it would lead to considering more rules as administrative in $\lambda_\mathtt{a}$. For example, consider an expression of the shape $e \equiv (\mathtt{rec}\ b_\mathsf{v}\ \mathtt{in}\,(\mathtt{rec}\ b\ \mathtt{in}\ e_1)\ e_2)$, with $\mathsf{dom}(b)\ \#\ \mathsf{FV}(b_\mathsf{v}) \cup \mathsf{FV}(e_2)$. This expression reduces by rules LIFT$_\circ$ and EM$_\circ$ to $e' \equiv (\mathtt{rec}\ b_\mathsf{v}, b\ \mathtt{in}\ e_1\ e_2)$. The purpose of performing the administrative reductions on the fly is to abstract over some reduction rules that are considered administrative in $\lambda_\mathtt{a}$, because they have no equivalent in $\lambda_\circ$. Here, rule LIFT$_\circ$ does have an equivalent in $\lambda_\circ$, so we may avoid the administrative reductions for b in $\lfloor e\rfloor^{\mathrm{TOP}}$ (and perform them for $\lfloor e'\rfloor^{\mathrm{TOP}}$).

Thus, for translating after b_v, we must define a translation function different from $\lfloor\cdot\rfloor^{\mathrm{TOP}}$, but nevertheless performing some administrative reductions. This is the purpose of the *allocating* translation $\lfloor\cdot\rfloor$. In fact, except for the `rec` case, $\lfloor\cdot\rfloor^{\mathrm{TOP}}$ and $\lfloor\cdot\rfloor$ perform exactly the same administrative reductions, and we define $\lfloor\cdot\rfloor^{\mathrm{TOP}}$ in terms of $\lfloor\cdot\rfloor$.

Standard In other the parts of e, where no administrative reductions are to be performed, we apply $[\![\cdot]\!]$.

$$
\begin{array}{lll}
\lfloor x \rfloor & \equiv \mathtt{Rec}\,\varepsilon\,\mathtt{in}\,x & \\
\lfloor \lambda x.e \rfloor & \equiv \mathtt{Rec}\,\ell = \lambda x.[\![e]\!]\,\mathtt{in}\,\ell & \\
\lfloor \{r\} \rfloor & \equiv \mathtt{Rec}\,\ell = \{r\}\,\mathtt{in}\,\ell & \\
\lfloor e\ v \rfloor & \equiv \mathtt{Rec}\,H_1,H_2\,\mathtt{in}\,E\ V & \text{if } \begin{cases} \lfloor e \rfloor \equiv \mathtt{Rec}\,H_1\,\mathtt{in}\,E \\ \lfloor v \rfloor \equiv \mathtt{Rec}\,H_2\,\mathtt{in}\,V \end{cases} \\
\lfloor e_1\ e_2 \rfloor & \equiv \mathtt{Rec}\,H\,\mathtt{in}\,[\![e_1]\!]\ E & \text{if } \begin{cases} e_2 \notin \mathsf{values} \\ \lfloor e_2 \rfloor \equiv \mathtt{Rec}\,H\,\mathtt{in}\,E \end{cases} \\
\lfloor e.X \rfloor & \equiv \mathtt{Rec}\,H\,\mathtt{in}\,E.X & \text{if } \lfloor e \rfloor \equiv \mathtt{Rec}\,H\,\mathtt{in}\,E \\
\lfloor \mathtt{rec}\ b\ \mathtt{in}\ e \rfloor & \equiv \mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![\mathtt{rec}\ b\ \mathtt{in}\ e]\!] &
\end{array}
$$

Fig. 24 The allocating translation from $\lambda_\circ$ to λ_a

5.1.2 The allocating translation

Let us now formally define $\lfloor\cdot\rfloor$. The idea is to translate the evaluated part of the input expression into a proper λ_a evaluation context, performing the administrative reductions on the fly. When the not-yet-evaluated parts of the expression are reached, the standard translation is used. For instance, given a function application $e_1\ e_2$, where e_2 is not a value, one can consider that the current evaluation point is inside e_2, and therefore that e_1 has remained untouched. So, we will use $[\![e_1]\!]$ and $\lfloor e_2 \rfloor$. The function $\lfloor\cdot\rfloor$ is defined in Figure 24.

Definition 26 (Locations and substitutions) We choose a set $\mathsf{Locs} \subseteq \mathsf{vars}$ of *locations*, ranged over by ℓ, such that Locs and $\mathsf{vars} \setminus \mathsf{Locs}$ are both infinite. We consider only $\lambda_\circ$ raw expressions whose free and bound variables are in $\mathsf{vars} \setminus \mathsf{Locs}$, which is from now on ranged over by x. We consider only λ_a raw configurations $\mathtt{Rec}\,H\,\mathtt{in}\,E$ such that $\mathsf{dom}(H) \subseteq \mathsf{Locs}$ and locations are never bound in E or the right-hand sides of H. From now on, we also call *substitutions*, ranged over by σ, functions from vars to vars whose support is disjoint from Locs. Composition of these substitutions is well defined as mere function composition. We call *variable allocations* such substitutions that are furthermore injective on their support and whose cosupport only contains locations. We denote them by ς (final sigma).

We stress in passing that the cosupport, and free variables of substitutions stay the same, e.g., cosupport may contain locations.

We then define $\lfloor\cdot\rfloor$ as a function from $\lambda_\circ$ equivalence classes to λ_a configurations (it is obviously well defined).

As for the standard translation, variables are translated into themselves. A function $\lambda x.e$ is translated to $\lambda x.[\![e]\!]$, but the result is allocated on the heap, at a fresh location ℓ: $\mathtt{Rec}\,\ell = \lambda x.[\![e]\!]\,\mathtt{in}\,\ell$. The translation of records is similar. For translating function application, we use a new notation: given two heaps H_1 and H_2 such that $\mathsf{dom}(H_1) \# \mathsf{dom}(H_2)$, we write H_1,H_2 for their concatenation, which is a heap again. If the argument part is not a value, then it is translated with $\lfloor\cdot\rfloor$, while the function is translated with $[\![\cdot]\!]$. If the argument is a value, then both parts are translated with $\lfloor\cdot\rfloor$. The translation of a record selection $e.X$ consists of translating e with $\lfloor\cdot\rfloor$ and then selecting the field X. Finally, a binding $\mathtt{rec}\ b\ \mathtt{in}\ e$ is translated to $\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![\mathtt{rec}\ b\ \mathtt{in}\ e]\!]$.

5.1.3 Generalized contexts

Given an expression e, in order to calculate $\lfloor e \rfloor^{\mathrm{TOP}}$, we will decompose e into its top-level binding b_1 and the rest of the expression e_1, and the result will be the translation of e_1, put in

some context representing b_1, written $\lfloor b_1 \rfloor^{\mathrm{TOP}}$, which is defined in Figure 27 (Section 5.1.8), using notions defined in Sections 5.1.3 to 5.1.7. The binding is divided into its evaluated part b_v and the rest b, which can be empty, but does not begin with a size-respecting definition. We start by giving an informal account of the handling of b_v and b, which leads us to the definition of a generalized notion of context in λ_a.

Let us first explain the translation of the unevaluated part b. In $[\![\cdot]\!]$, the Dummy function produces instructions for allocating dummy blocks. In the top-level translation, these blocks are directly allocated by the function TDum (see Section 5.1.8), which returns the heap of dummy blocks and the substitution replacing variables with the corresponding locations. As a first example, given a binding $b \equiv (x_1 =_{[?]} e_1, x_2 =_{[n]} e_2)$, $\mathsf{TDum}(b)$ essentially returns a heap $\ell_2 = \texttt{alloc}\ n$ and the substitution $[x_2 \mapsto \ell_2]$. This corresponds to the fact that after the pre-allocation pass (as generated by the standard translation), the update pass takes place under this heap and substitution.

In $[\![\cdot]\!]$, the Update function produces instructions to either update a dummy block with the translation of the definition, or to perform the binding implied by the definition. In $\lfloor\cdot\rfloor^{\mathrm{TOP}}$, the only difference is that the first definition in b is translated with $\lfloor\cdot\rfloor$, while the remaining ones – still considered to lie past the current evaluation point – are translated with $[\![\cdot]\!]$. This is done by function TUp (see Section 5.1.8). On the previous example, if $\lfloor e_1 \rfloor \equiv \texttt{Rec}\, H_1\, \texttt{in}\, E_1$, then $\mathsf{TUp}(b)$ essentially returns the heap H_1 and the binding $x_1 = E_1, \_ = \texttt{update}\ x_2\ [\![e_2]\!]$. Under the substitution returned by TDum, the second definition becomes $\texttt{update}\ \ell_2\ [\![e_2]\!]$, as expected.

Now, what should be the top-level translation, written $\mathsf{Top}(b_\mathsf{v})$, of the evaluated part b_v? As mentioned above, this translation yields a heap and a substitution. The translation of definitions is relatively natural, but it is difficult to assemble the results in a coherent manner. First, consider a single definition $x \diamond v$. The allocating translation of v is an answer, of the shape $\texttt{Rec}\, H\, \texttt{in}\, V$. It is thus clear that the generated heap and substitution should be H and $[x \mapsto V]$, respectively.

The next question is how to assemble the results obtained for each definition. First, we remark that in the absence of forward references, substitutions should be composed from right to left. For instance, on a binding like $b_\mathsf{v} \equiv (x_1 =_{[?]} x_0, x_2 =_{[?]} x_1)$, the generated substitution must be $[x_1 \mapsto x_0] \circ [x_2 \mapsto x_1]$, and not the converse. Thus, definitions can be altered by previous definitions, which may have replaced some variables with other values.

However, because of forward references in $\lambda_\circ$ bindings, the translated definitions may also have to be altered by subsequent definitions. For instance, consider the binding $b_\mathsf{v} \equiv (x_1 =_{[?]} x_2, x_2 =_{[n]} \lambda x.x)$, where $n = \mathsf{size}(\lambda x.x)$. The top-level translation turns b_v into a heap and a substitution. The translation of the first definition consists of the heap $H_1 \equiv \varepsilon$ and the substitution $\sigma = [x_1 \mapsto x_2]$, so that subsequent occurrences of x_1 are replaced with x_2. Then, we translate the second definition. This gives $H_2 \equiv (\ell_2 = \lambda x.x)$ and $\varsigma = [x_2 \mapsto \ell_2]$, for some fresh location ℓ_2. Naively, one could think that the substitution corresponding to the whole binding should be the right-to-left composition of the obtained substitutions. But this is wrong, since the obtained substitution would be $\sigma \circ \varsigma$. Under this substitution, a call to x_1 becomes $[x_1 \mapsto x_2]([x_2 \mapsto \ell_2](x_1)) = x_2$, while it should rather be directed to ℓ_2. This example illustrates that variable allocations performed by the translation are expected to alter previous forward references to them, which possibly appear as substitutions.

This leads us to define a new notion of context in λ_a, called *generalized context*, in terms of which we define the translation of bindings. The functions TDum, TUp, and Top will be defined as returning generalized contexts, which makes their uniform treatment easier. Basically, the idea of generalized contexts is that they contain a heap, an allocation context, and two substitutions, rather than one. This allows distinguishing variable allocations $x \mapsto \ell$,

which might alter previous translations, from normal substitutions $x \mapsto y$, which may not. Basically, only definitions of known size can alter previous translations, because they are the only ones that can be forward referenced. Furthermore, crucially, we will require that the normal substitution of a generalized context be *one-way*, in the following sense.

Definition 27 (One-way substitution) A substitution σ (with implicitly $\mathsf{supp}(\sigma) \# \mathsf{Locs}$, by Definition 26) is *one-way* iff $\mathsf{supp}(\sigma) \# \mathsf{cosupp}(\sigma)$.

From the informal explanations above, it should sound natural that the unknown size definitions of the shape $x =_{[?]} y$ generate one-way substitutions. Indeed, they only "go left" in the binding, and no binding may define, say $x =_{[?]} y, y =_{[?]} x$, because of the syntactic restriction on forward references.

Let us first prove the following easy lemmas on substitutions.

Lemma 28 *For all one-way substitutions σ, $\sigma \circ \sigma = \sigma$.*

Proof For all variable x, either $x \notin \mathsf{supp}(\sigma)$, and then both sides are equal to x, or $x \in \mathsf{supp}(\sigma)$, but then $\sigma(x) \in \mathsf{cosupp}(\sigma)$, which by hypothesis implies $\sigma(x) \notin \mathsf{supp}(\sigma)$, hence $\sigma(\sigma(x)) = \sigma(x)$, as expected. □

Lemma 29 *For all substitutions σ_1 and σ_2,*

- $\mathsf{supp}(\sigma_1 \circ \sigma_2) \subseteq \mathsf{supp}(\sigma_1) \cup \mathsf{supp}(\sigma_2)$, *and*
- $\mathsf{cosupp}(\sigma_1 \circ \sigma_2) \subseteq \mathsf{cosupp}(\sigma_1) \cup \mathsf{cosupp}(\sigma_2)$.

Proof The first point is point is easy by contradiction.

For the second point, assume $x \in \mathsf{cosupp}(\sigma_1 \circ \sigma_2)$. There is some $y \neq x$ such that $(\sigma_1 \circ \sigma_2)(y) = x$.

Let $z = \sigma_2(y)$. If $x \notin \mathsf{cosupp}(\sigma_1)$, then $z = x$, so $\sigma_2(y) = x$, and $x \in \mathsf{cosupp}(\sigma_2)$.

□

Lemma 30 *For all substitutions σ_1 and σ_2, if $\mathsf{FV}(\sigma_1) \# \mathsf{supp}(\sigma_2)$, then $\sigma_1 \circ \sigma_2 = \sigma_1(\sigma_2) \circ \sigma_1$.*

Proof Let $x \in \mathsf{vars}$.

- If $x \in \mathsf{supp}(\sigma_2)$, then $x \notin \mathsf{FV}(\sigma_1)$, so $(\sigma_1(\sigma_2) \circ \sigma_1)(x) = (\sigma_1(\sigma_2))(x)$, which is the expected result.
- Otherwise, if $x \in \mathsf{supp}(\sigma_1)$, then $\sigma_1(x) \in \mathsf{cosupp}(\sigma_1)$, so $\sigma_1(x) \notin \mathsf{supp}(\sigma_2)$, hence $(\sigma_1(\sigma_2) \circ \sigma_1)(x) = \sigma_1(x) = (\sigma_1 \circ \sigma_2)(x)$.

□

Lemma 31 *For all σ, ς, if $\mathsf{supp}(\sigma) \# \mathsf{supp}(\varsigma)$, then $\varsigma(\sigma) \circ \varsigma = \varsigma \circ \sigma \circ \varsigma$.*

Proof Let $x \in \mathsf{vars}$.

- If $\varsigma(x) \in \mathsf{supp}(\sigma)$, then both sides are equal to $(\varsigma \circ \sigma \circ \varsigma)(x)$.
- Otherwise, since ς is one-way, $\varsigma \circ \varsigma = \varsigma$, so $(\varsigma(\sigma) \circ \varsigma)(x) = \varsigma(x) = (\varsigma \circ \varsigma)(x) = (\varsigma \circ \sigma \circ \varsigma)(x)$.

□

Corollary 32 *For all ς, σ, if $\mathsf{supp}(\varsigma) \# \mathsf{supp}(\sigma)$, then $\varsigma \circ \sigma = \varsigma \circ \sigma \circ \varsigma$.*

Proof By Lemmas 30 and 31, since obviously $\mathsf{supp}(\varsigma) \# \mathsf{supp}(\sigma)$ implies $\mathsf{FV}(\varsigma) \# \mathsf{supp}(\sigma)$. □

Then, we have the following obvious, but useful result.

Proposition 33 *For all α, σ, E, if $\mathsf{FV}(\sigma) \# \mathsf{Capt}_{\square}(\alpha)$, then $\sigma(\alpha[E]) \equiv (\sigma(\alpha))[\sigma(E)]$.*

Corollary 34 *For all α, σ, E, if $\sigma \circ \sigma = \sigma$ and $\mathsf{FV}(\sigma) \# \mathsf{Capt}_{\square}(\alpha)$, then $\sigma(\alpha[E]) \equiv \sigma(\alpha[\sigma(E)])$.*

Proof By Proposition 33, $\sigma(\alpha[E]) \equiv (\sigma(\alpha))[\sigma(E)] \equiv (\sigma(\alpha))[(\sigma \circ \sigma)(E)] \equiv \sigma(\alpha[\sigma(E)])$. □

Then, we give the following sufficient condition for the composition of two one-way substitutions to be one-way. We recall that $\mathsf{FV}(\sigma_1) \# \mathsf{supp}(\sigma_2)$ means

- $\mathsf{supp}(\sigma_1) \# \mathsf{supp}(\sigma_2)$ and
- $\mathsf{cosupp}(\sigma_1) \# \mathsf{supp}(\sigma_2)$.

Lemma 35 *For all one-way substitutions σ_1 and σ_2, if $\mathsf{FV}(\sigma_1) \# \mathsf{supp}(\sigma_2)$, then*

- *$\sigma_1 \circ \sigma_2$ is one-way,*
- $\mathsf{supp}(\sigma_1 \circ \sigma_2) = \mathsf{supp}(\sigma_1) \cup \mathsf{supp}(\sigma_2)$,
- $\mathsf{cosupp}(\sigma_1) \subseteq \mathsf{cosupp}(\sigma_1 \circ \sigma_2)$.

Proof We prove that $\sigma_1 \circ \sigma_2$ is one-way by contradition. Let $\sigma = \sigma_1 \circ \sigma_2$ and assume the existence of $x \in \mathsf{cosupp}(\sigma) \cap \mathsf{supp}(\sigma)$, i.e., the existence of x, y, and z, such that

- $\sigma(x) = y$ with $x \neq y$, and
- $\sigma(z) = x$ with $x \neq z$.

Let then $x' = \sigma_2(x)$ and $z' = \sigma_2(z)$, so that we informally have:

$$x \xrightarrow{\sigma_2} x' \xrightarrow{\sigma_1} y$$

$$z \xrightarrow{\sigma_2} z' \xrightarrow{\sigma_1} x$$

- If $x \neq x'$ and $x \neq z'$, then $x \in \mathsf{supp}(\sigma_2) \cap \mathsf{cosupp}(\sigma_1)$ which is impossible by hypothesis.
- If $x = x'$ and $x \neq z'$, then $x \in \mathsf{supp}(\sigma_1) \cap \mathsf{cosupp}(\sigma_1)$, which is impossible because σ_1 is one-way.
- If $x = z'$ and $x \neq x'$, then $x \in \mathsf{supp}(\sigma_2) \cap \mathsf{cosupp}(\sigma_2)$, which is impossible because σ_2 is one-way.
- If $x = x'$ and $x = z'$, then $\sigma(z) = y$ which is impossible since $\sigma(z) = x$ and $x \neq y$.

The second point is proved as follows.

- By Lemma 29, $\mathsf{supp}(\sigma_1 \circ \sigma_2) \subseteq \mathsf{supp}(\sigma_1) \cup \mathsf{supp}(\sigma_2)$;
- If $x \in \mathsf{supp}(\sigma_1) \cup \mathsf{supp}(\sigma_2)$ but $x \notin \mathsf{supp}(\sigma_1 \circ \sigma_2)$, i.e., $\sigma_1(\sigma_2(x)) = x$, then let $y = \sigma_2(x)$. If $x = y$, then $\sigma_1(x) = \sigma_1(y) = x$, so x is neither in $\mathsf{supp}(\sigma_1)$ nor in $\mathsf{supp}(\sigma_2)$, which contradicts $x \in \mathsf{supp}(\sigma_1) \cup \mathsf{supp}(\sigma_2)$. Otherwise, we have $y \neq x$, which implies $x \in \mathsf{supp}(\sigma_2)$, and furthermore and $(\sigma_1 \circ \sigma_2)(x) = \sigma_1(y) = x$, so $x \in \mathsf{cosupp}(\sigma_1) \cap \mathsf{supp}(\sigma_2)$, a contradiction.

$$\mathsf{FV}(\langle H\,;\alpha\,;\sigma\,;\varsigma\rangle) = (\mathsf{FV}(H)\cup\mathsf{FV}(\alpha)\cup\mathsf{FV}(\sigma)\cup\mathsf{FV}(\varsigma))\setminus\mathsf{dom}(H)$$
$$\mathsf{Capt}_{\square}(\langle H\,;\alpha\,;\sigma\,;\varsigma\rangle) = \mathsf{Capt}_{\square}(\alpha)\cup\mathsf{supp}(\sigma)\cup\mathsf{supp}(\varsigma)$$

Fig. 25 Free variables and captured variables for generalized contexts

$$\frac{H\equiv H' \qquad \alpha\equiv\alpha' \qquad \sigma\equiv\sigma' \qquad \varsigma\equiv\varsigma'}{\langle H\,;\alpha\,;\sigma\,;\varsigma\rangle\equiv\langle H'\,;\alpha'\,;\sigma'\,;\varsigma'\rangle} \qquad \frac{\ell'\notin\mathsf{dom}(H) \qquad \sigma'=[\ell\mapsto\ell']\sigma \qquad \varsigma'=[\ell\mapsto\ell']\varsigma}{\langle(H,\ell=S)\,;\alpha\,;\sigma\,;\varsigma\rangle\equiv\langle(H,\ell'=S)\,;\alpha\,;\sigma'\,;\varsigma'\rangle}$$

Fig. 26 Structural equivalence of generalized contexts

Let us now prove the third point: let $x\in\mathsf{cosupp}(\sigma_1)$. There exists $y\neq x$ such that $\sigma_1(y)=x$. By hypothesis, this y is not in $\mathsf{supp}(\sigma_2)$, so $(\sigma_1\circ\sigma_2)(y)=x$, and $x\in\mathsf{cosupp}(\sigma_1\circ\sigma_2)$. □

Definition 36 (Generalized contexts) A *generalized context* is a 4-tuple of a heap H, an allocation context α, a substitution σ, and a variable allocation ς, written $\gamma ::= \langle H\,;\alpha\,;\sigma\,;\varsigma\rangle$, such that

- σ is one-way,
- $\mathsf{supp}(\sigma)\#\mathsf{supp}(\varsigma)$,
- α and the range of H do not have any free location,
- $\mathsf{cosupp}(\sigma)\cap\mathsf{Locs}\subseteq\mathsf{dom}(H)$,
- and $\mathsf{cosupp}(\varsigma)\subseteq\mathsf{dom}(H)$.

A *generalized evaluation context* is a generalized context whose allocation context is an evaluation context, i.e., a generalized context of the shape $\langle H\,;\xi\,;\sigma\,;\varsigma\rangle$.

The generalized contexts generated by the translation of size-respecting bindings will have $\square$ as their allocation context. We call such generalized contexts *generalized bindings*, and write them β.

The generalized contexts generated by dummy allocation of bindings will have the shape $\langle H\,;\square\,;\mathsf{id}\,;\varsigma\rangle$. We call such generalized contexts *generalized dummy allocations*, and write them δ.

Also, we define the syntactic sugar $\langle H\,;B\,;\sigma\,;\varsigma\rangle$, which, if B is not empty, denotes $\langle H\,;\mathtt{let}\,B\,\mathtt{in}\,\square\,;\sigma\,;\varsigma\rangle$, and otherwise denotes $\langle H\,;\square\,;\sigma\,;\varsigma\rangle$. Further, bindings B are implicitly coerced to generalized contexts $\langle\varepsilon\,;B\,;\mathsf{id}\,;\mathsf{id}\rangle$. Finally, we simply write σ for $\langle\varepsilon\,;\square\,;\sigma\,;\mathsf{id}\rangle$, and define $\mathsf{Subst}(\langle H\,;\alpha\,;\sigma\,;\varsigma\rangle)=\varsigma\circ\sigma$ and $\mathsf{Cont}(\langle H\,;\alpha\,;\sigma\,;\varsigma\rangle)\equiv\alpha$.

Notes: $\mathsf{dom}(H)$ contains only locations, and is thus inherently disjoint from $\mathsf{supp}(\sigma)$ and $\mathsf{supp}(\varsigma)$. Also, recall that every evaluation context ξ is also an allocation context α.

Next, we define structural equivalence on generalized contexts, using the definition of free variables in Figure 25. For helping the intuition we also define the captured variables for generalized contexts. The intuition behind structural equivalence of generalized contexts is that the locations bound in $\mathsf{dom}(H)$ may be renamed freely, since they may only be mentioned in the cosupport of σ and ς. Formally, structural equivalence is defined in Figure 26, as the least equivalence relation respecting the rules. The first rule says that α-equivalence on expressions, heaps, and stored values is included; the second rule says that a location in the heap may be renamed, provided it does not clash with another one.

Example 37 Consider the bindings $b_{\mathsf{v}}\equiv(x_1=_{[?]}x_0, x_2=_{[?]}x_4, x_3=_{[?]}x_1, x_4=_{[n]}\lambda x.x)$, which is an interleaving of previous examples, and $b\equiv(x_5=_{[n]}x_2)$.

Via Top, each definition in b_{v} yields a generalized context:

- x_1 yields $\gamma_{11} \equiv \langle \varepsilon \,;\, \Box \,;\, [x_1 \mapsto x_0] \,;\, \mathsf{id} \rangle$,
- x_2 yields $\gamma_{12} \equiv \langle \varepsilon \,;\, \Box \,;\, [x_2 \mapsto x_4] \,;\, \mathsf{id} \rangle$,
- x_3 yields $\gamma_{13} \equiv \langle \varepsilon \,;\, \Box \,;\, [x_3 \mapsto x_1] \,;\, \mathsf{id} \rangle$, and
- x_4 yields $\gamma_{14} \equiv \langle \ell = \lambda x.x \,;\, \Box \,;\, \mathsf{id} \,;\, [x_4 \mapsto \ell] \rangle$.

The not yet evaluated binding b yields the heap $H \equiv \ell' = \texttt{alloc}\ n$ and the variable allocation $\varsigma = [x_5 \mapsto \ell']$ by function TDum, which we write $\gamma_2 \equiv \langle H \,;\, \Box \,;\, \mathsf{id} \,;\, \varsigma \rangle$.

Via TUp, b yields the heap $H' \equiv \varepsilon$ and the binding $B \equiv (\_ = \texttt{update}\ x_5\ x_2)$, which we write $\gamma_3 \equiv \langle H' \,;\, \texttt{let}\ B\ \texttt{in}\ \Box \,;\, \mathsf{id} \,;\, \mathsf{id} \rangle$. Note that this is the only use of generalized contexts using allocation contexts (here $\texttt{let}\ B\ \texttt{in}\ \Box$) which are not evaluation contexts.

5.1.4 Composition of generalized contexts

We then need a notion of composition of generalized contexts, in order to assemble the pieces of our translation. The guiding intuition here is that when composing two generalized contexts $\gamma_1 \equiv \langle H_1 \,;\, \alpha_1 \,;\, \sigma_1 \,;\, \varsigma_1 \rangle$ and $\gamma_2 \equiv \langle H_2 \,;\, \alpha_2 \,;\, \sigma_2 \,;\, \varsigma_2 \rangle$, we want the result to be well-defined and equal to

$$\langle H_1, H_2 \,;\, \alpha_1[\alpha_2] \,;\, \sigma_1 \circ \sigma_2 \,;\, \varsigma_1 + \varsigma_2 \rangle,$$

but we also want the following two equations to hold for any composable γ_1 and γ_2, and for any expression E:

$$\begin{aligned}((\varsigma_1 + \varsigma_2) \circ \sigma_1 \circ \sigma_2)(H_1, H_2) \equiv\ & ((\varsigma_1 + \varsigma_2) \circ \sigma_1)(H_1), \\ & (\varsigma_1 \circ \varsigma_2(\sigma_1) \circ \varsigma_2 \circ \sigma_2)(H_2))\end{aligned}$$

and

$$((\varsigma_1 + \varsigma_2) \circ \sigma_1 \circ \sigma_2)(\alpha_1[\alpha_2[E]]) \equiv ((\varsigma_1 + \varsigma_2) \circ \sigma_1)(\alpha_1[(\varsigma_2 \circ \sigma_2)(\alpha_2[E])]).$$

In these equations, the left member is (part of) what we get by applying the result of the composition to E, as defined below (Definition 45). The right member describes how we would like the four substitutions to interact. For instance, σ_2 is a standard substitution, which does not affect upper levels of context: in both left members, it does not act on the components of γ_1. On the other hand ς_2 has to affect them, but should not be shortcut by α_1 and σ_1, which explains why we require that it still affects the variables in $\alpha_2[E]$ and H_2 before, respectively, α_1 and σ_1, which come first in the left members.

We use the following definition, which natural except for the domain of definition: two generalized contexts $\gamma_1 \equiv \langle H_1 \,;\, \alpha_1 \,;\, \sigma_1 \,;\, \varsigma_1 \rangle$ and $\gamma_2 \equiv \langle H_2 \,;\, \alpha_2 \,;\, \sigma_2 \,;\, \varsigma_2 \rangle$ are *composable*, written $\gamma_1 \succ \gamma_2$, iff

- the four substitutions $\sigma_1, \varsigma_1, \sigma_2$, and ς_2 have pairwise disjoint supports,
- $\mathsf{supp}(\sigma_2) \# \mathsf{FV}(\gamma_1)$,
- $\mathsf{Capt}_\Box(\alpha_1) \# \mathsf{FV}(\sigma_2) \cup \mathsf{FV}(\varsigma_2)$.

These conditions are oviously preserved by structural equivalence, which justifies the definition of composability on equivalence classes of generalized contexts.

We then state:

Definition 38 (Composition of generalized contexts) For all such composable generalized contexts γ_1 and γ_2 define their *composition* $\gamma_1 \circledast \gamma_2$ by $\gamma_1 \circledast \gamma_2 \equiv \langle H_1, H_2 \,;\, \alpha_1[\alpha_2] \,;\, \sigma_1 \circ \sigma_2 \,;\, \varsigma_1 + \varsigma_2 \rangle$, provided $\mathsf{dom}(H_1) \# \mathsf{dom}(H_2)$ (which can always be reached by structural equivalence).

Proposition 39 *The conditions for being composable are equivalent to*

- $\mathsf{FV}(\sigma_1) \# \mathsf{supp}(\sigma_2)$,
- $\mathsf{supp}(\varsigma_1) \# \mathsf{supp}(\varsigma_2)$,
- $\mathsf{supp}(\sigma_1) \cup \mathsf{supp}(\sigma_2) \# \mathsf{supp}(\varsigma_1) \cup \mathsf{supp}(\varsigma_2)$,
- $\mathsf{supp}(\sigma_2) \# \mathsf{FV}(H_1) \cup \mathsf{FV}(\alpha_1)$,
- $\mathsf{cosupp}(\sigma_2) \# \mathsf{Capt}_{\Box}(\alpha_1)$,
- $\mathsf{supp}(\varsigma_2) \# \mathsf{supp}(\sigma_1) \cup \mathsf{Capt}_{\Box}(\alpha_1)$.

Proof Easy check. □

In the light of this, the conditions for composability may be understood as follows:

- The first three items ensure that the result is a well-formed generalized context. Well, actually they do a bit more: they use the sufficient condition of Lemma 35, requiring $\mathsf{FV}(\sigma_1) \# \mathsf{supp}(\sigma_2)$ and $\mathsf{supp}(\sigma_1) \cup \mathsf{supp}(\sigma_2) \# \mathsf{supp}(\varsigma_1) \cup \mathsf{supp}(\varsigma_2)$ instead of requiring $\sigma_1 \circ \sigma_2$ to be one-way, and $\mathsf{supp}(\sigma_1 \circ \sigma_2) \# \mathsf{supp}(\varsigma_1) \cup \mathsf{supp}(\varsigma_2)$ to hold. But this more restrictive requirement allows an easy proof of weak associativity for composition of generalized contexts, and is general enough for our purposes.
- The fourth item ensures that σ_2 does not affect H_1 and α_1.
- The fifth item ensures that σ_2 is not shortcut by α_1 (i.e., $\sigma_2(\alpha_1[\ldots]) \equiv \sigma_2(\alpha_1)[\sigma_2(\ldots)]$, which by the previous point is in fact $\alpha_1[\sigma_2(\ldots)]$).
- The sixth item ensures that ς_2 is not shortcut by σ_1 and α_1.

Example 40 Consider again b_{v} from Example 37. Its top-level translation is $\gamma_{11} \circledast \gamma_{12} \circledast \gamma_{13} \circledast \gamma_{14}$, which is exactly $\gamma_1 \equiv \langle H_{b_{\mathsf{v}}} \,;\, \Box \,;\, \sigma_{b_{\mathsf{v}}} \,;\, \varsigma_{b_{\mathsf{v}}} \rangle$, with the heap $H_{b_{\mathsf{v}}} \equiv \ell \mathrel{=} \lambda x.x$, the variable allocation $\varsigma_{b_{\mathsf{v}}} = [x_4 \mapsto \ell]$, and the substitution $\sigma_{b_{\mathsf{v}}} = [x_1, x_3 \mapsto x_0, x_2 \mapsto x_4]$. Note that the rest of the translation ensures that variable allocations are always applied after substitutions, so that x_2 will eventually be redirected to ℓ.

We noww prove useful sufficient conditions for composability and associativity. They use the following notation for, respectively, the *unknown size* and *known size* variables of a binding b:

- $\mathsf{UV}(b) = \{x \mid \exists e, (x \mathrel{=}_{[?]} e) \in b\}$,
- $\mathsf{KV}(b) = \{x \mid \exists n, e, (x \mathrel{=}_{[n]} e) \in b\}$.

Proposition 41 *If (b_1, b_2) is syntactically correct, then $\mathsf{FV}(b_1) \# \mathsf{UV}(b_2)$.*

Definition 42 For all generalized contexts $\gamma \equiv \langle H \,;\, \alpha \,;\, \sigma \,;\, \varsigma \rangle$, and correct bindings b, we say that *b justifies γ*, and write $b \vdash \gamma$, iff:

- $\mathsf{FV}(\gamma) \subseteq \mathsf{FV}(b)$, and more specifically,
- $\mathsf{supp}(\sigma) \subseteq \mathsf{UV}(b)$,
- $\mathsf{supp}(\varsigma) \subseteq \mathsf{KV}(b)$.

Lemma 43 *Assume a correct binding of the shape (b_1, b_2) and two generalized contexts $\gamma_i \equiv \langle H_i \,;\, \alpha_i \,;\, \sigma_i \,;\, \varsigma_i \rangle$, such that $b_i \vdash \gamma_i$, for $i = 1, 2$. If moreover $\mathsf{Capt}_{\Box}(\alpha_1) = \emptyset$, then $\gamma_1 \succ \gamma_2$ and $b_1, b_2 \vdash \gamma_1 \circledast \gamma_2$.*

Proof First, the four involved substitutions have as supports the domains of pairwise disjoint parts of b_1, b_2, hence have pairwise disjoint supports. Furthermore, since b_1, b_2 is correct, b_1

makes no (forward) reference to $\mathsf{UV}(b_2)$, hence $\mathsf{supp}(\sigma_2) \# \mathsf{FV}(\gamma_1)$. Thus, $\mathsf{Capt}_\Box(\alpha_1)$ being empty, we have $\gamma_1 \succ \gamma_2$.

Furthermore, we have

$$\mathsf{FV}(\gamma_1 \circledast \gamma_2) \subseteq \mathsf{FV}(\gamma_1) \cup \mathsf{FV}(\gamma_2) \subseteq \mathsf{FV}(b_1) \cup \mathsf{FV}(b_2) = \mathsf{FV}(b_1, b_2).$$

By a similar reasoning on substitutions, we obtain $b_1, b_2 \vdash \gamma_1 \circledast \gamma_2$ as desired. □

Lemma 44 *Assume a correct binding of the shape (b_1, b_2, b_3) and three generalized contexts $\gamma_i \equiv \langle H_i \,;\, \alpha_i \,;\, \sigma_i \,;\, \varsigma_i \rangle$, such that $b_i \vdash \gamma_i$, for $i = 1, 2, 3$. If moreover $\mathsf{Capt}_\Box(\alpha_i) = \emptyset$, for $i = 1, 2$, then $(\gamma_1 \circledast \gamma_2) \circledast \gamma_3$ and $\gamma_1 \circledast (\gamma_2 \circledast \gamma_3)$ are defined and equal.*

Proof By the previous Lemma, we obtain $\gamma_1 \succ \gamma_2$ and $b_1, b_2 \vdash \gamma_1, \gamma_2$, hence by the same Lemma, $(\gamma_1 \circledast \gamma_2) \succ \gamma_3$. Symmetrically, $\gamma_1 \succ (\gamma_2 \circledast \gamma_3)$. Thus, both sides are defined at the same time. Equality is then a simple check. □

5.1.5 Generalized context application

We have seen that the top-level binding will be translated as a generalized context. We will then fill the context hole with the translation of the rest of the expression, using generalized context application, which we now define.

Definition 45 (Generalized context application) For every generalized context $\gamma \equiv \langle H \,;\, \alpha \,;\, \sigma \,;\, \varsigma \rangle$ and configuration $C \equiv \mathtt{Rec}\, H' \,\mathtt{in}\, E$, let $\gamma[C] \equiv (\mathtt{Rec}\, (\varsigma \circ \sigma)(H, H') \,\mathtt{in}\, (\varsigma \circ \sigma)(\alpha[E]))$ be the *application* of γ to C, provided $\mathsf{dom}(H') \# \mathsf{dom}(H) \cup \mathsf{cosupp}(\sigma) \cup \mathsf{cosupp}(\varsigma)$ (which may always be reached by structural equivalence).

Example 46 Consider again the binding (b_v, b) from Example 37. Its translation is $\gamma_2 \circledast \gamma_1 \circledast \gamma_3$, which is exactly $\gamma \equiv \langle H_0 \,;\, \alpha_0 \,;\, \sigma_{b_\mathsf{v}} \,;\, \varsigma \circ \varsigma_{b_\mathsf{v}} \rangle$, with

- the heap $H_0 \equiv \begin{pmatrix} \ell = \lambda x.x, \\ \ell' = \mathtt{alloc}\ n \end{pmatrix}$
- and the context $\alpha_0 \equiv \mathtt{let}\ B\ \mathtt{in}\ \Box$.

If, for instance, γ is filled with a configuration $\mathtt{Rec}\, H \,\mathtt{in}\, E$, if the conditions for the generalized context application are met, we get $\gamma[\mathtt{Rec}\, H \,\mathtt{in}\, E] \equiv \mathtt{Rec}\, \sigma(H_0, H) \,\mathtt{in}\, \sigma(\alpha_0[E])$, where $\sigma = (\varsigma \circ \varsigma_{b_\mathsf{v}} \circ \sigma_{b_\mathsf{v}}) = \begin{pmatrix} x_5 \mapsto \ell', \\ x_1, x_3 \mapsto x_0, \\ x_2 \mapsto \ell, \\ x_4 \mapsto \ell \end{pmatrix}$.

We now prove the equations that had motivated the definition of generalized context composition.

Lemma 47 *For all composable generalized contexts $\gamma_i \equiv \langle H_i \,;\, \alpha_i \,;\, \sigma_i \,;\, \varsigma_i \rangle$ and configuration $\mathtt{Rec}\, H \,\mathtt{in}\, E$, if $\mathsf{dom}(H) \# \mathsf{dom}(H_1, H_2)$, then*

$$\begin{aligned}(\gamma_1 \circledast \gamma_2)[\mathtt{Rec}\, H \,\mathtt{in}\, E] \equiv \mathtt{Rec}\ & ((\varsigma_1 + \varsigma_2) \circ \sigma_1)(H_1), \\ & (\varsigma_1 \circ \varsigma_2(\sigma_1) \circ \varsigma_2 \circ \sigma_2)(H_2, H) \\ & \mathtt{in}\, ((\varsigma_1 + \varsigma_2) \circ \sigma_1)(\alpha_1[(\varsigma_2 \circ \sigma_2)(\alpha_2[E])]).\end{aligned}$$

Proof First, by $\gamma_1 \succ \gamma_2$, we have $\mathsf{FV}(H_1) \# \mathsf{supp}(\sigma_2)$, so $((\varsigma_1 + \varsigma_2) \circ \sigma_1 \circ \sigma_2)(H_1) \equiv ((\varsigma_1 + \varsigma_2) \circ \sigma_1)(H_1)$.

Furthermore, $\varsigma_1 + \varsigma_2 = \varsigma_1 \circ \varsigma_2$. But by $\gamma_1 \succ \gamma_2$ again, we have $\mathsf{supp}(\varsigma_2) \# \mathsf{supp}(\sigma_1)$. So by Lemma 30, $\varsigma_2 \circ \sigma_1 = \varsigma_2(\sigma_1) \circ \varsigma_2$. This gives $((\varsigma_1 + \varsigma_2) \circ \sigma_1 \circ \sigma_2)(H_2, H) \equiv (\varsigma_1 \circ \varsigma_2(\sigma_1) \circ \varsigma_2 \circ \sigma_2)(H_2, H)$.

Now, by composability again, $\mathsf{Capt}_\Box(\alpha_1) \# \mathsf{FV}(\sigma_2) \cup \mathsf{FV}(\varsigma_2)$. But $\mathsf{FV}(\varsigma_2 \circ \sigma_2) \subseteq \mathsf{FV}(\sigma_2) \cup \mathsf{FV}(\varsigma_2)$, so $\mathsf{Capt}_\Box(\alpha_1) \# \mathsf{FV}(\varsigma_2 \circ \sigma_2)$. By Proposition 33 and the above, this yields $(\varsigma_2 \circ \sigma_2)(\alpha_1[\alpha_2[E]]) \equiv ((\varsigma_2 \circ \sigma_2)\alpha_1)[(\varsigma_2 \circ \sigma_2)(\alpha_2[E])]$.

But $\mathsf{supp}(\sigma_2) \# \mathsf{FV}(\alpha_1)$ and $\varsigma_2 \circ \varsigma_2 = \varsigma_2$, so $((\varsigma_2 \circ \sigma_2)\alpha_1)[(\varsigma_2 \circ \sigma_2)(\alpha_2[E])] = (\varsigma_2(\alpha_1))[(\varsigma_2 \circ \varsigma_2 \circ \sigma_2)(\alpha_2[E])]$, hence by Proposition 33 again, this is equal to $\varsigma_2(\alpha_1[(\varsigma_2 \circ \sigma_2)(\alpha_2[E])])$. Thus, $((\varsigma_1 + \varsigma_2) \circ \sigma_1 \circ \sigma_2)(\alpha_1[\alpha_2[E]])$ is indeed equal to $(\varsigma_1 \circ \varsigma_2(\sigma_1) \circ \varsigma_2)(\alpha_1[(\varsigma_2 \circ \sigma_2)(\alpha_2[E])])$, which is in turn equal to $((\varsigma_1 + \varsigma_2) \circ \sigma_1)(\alpha_1[(\varsigma_2 \circ \sigma_2)(\alpha_2[E])])$, as desired. □

5.1.6 Weak composition of generalized contexts

Although the notion of composition of generalized contexts is needed to properly translate size-respecting bindings, it is somewhat inconvenient to reason with. For instance, the usual equation $(\gamma_1 \circledast \gamma_2)[C] \equiv \gamma_1[\gamma_2[C]]$ obviously does not hold in general: the variable allocation of γ_2 may affect γ_1 in $(\gamma_1 \circledast \gamma_2)[C]$, but not in $\gamma_1[\gamma_2[C]]$. Nevertheless, when γ_1 and γ_2 stem from distinct bindings with no defined variable in common, we recover more standard properties. More generally, we define the following notions of *context interference* and *weak composition*, which take advantage of such cases in the following sense: weak composition has more standard properties than $\circledast$, and coincides with it when the considered contexts do not interfere. We first define weak composition and show that it satisfies the equation above.

Definition 48 (Weak composition of generalized contexts) Given $\gamma_i \equiv \langle H_i \,;\, \alpha_i \,;\, \sigma_i \,;\, \varsigma_i \rangle$, for $i = 1, 2$, if $\gamma_1 \succ \gamma_2$ is defined, we define $\gamma_1 \circ \gamma_2 \equiv \langle (H_1, H_2) \,;\, \alpha_1[\alpha_2] \,;\, (\varsigma_1 \circ \sigma_1 \circ \varsigma_2 \circ \sigma_2) \,;\, \mathrm{id} \rangle$.

Proposition 49 *For all γ_1, γ_2, and C, $(\gamma_1 \circ \gamma_2)[C] \equiv \gamma_1[\gamma_2[C]]$, when the former is defined.*

Proof By definition of weak composition and generalized context application. □

Now, we define context interference, and state the expected result.

Definition 50 (Context interference) Given two generalized contexts $\gamma_i \equiv \langle H_i \,;\, \alpha_i \,;\, \sigma_i \,;\, \varsigma_i \rangle$, for $i = 1, 2$, let us say that the pair (the order matters) (γ_1, γ_2) *interferes* iff $\mathsf{supp}(\varsigma_2)$ intersects $\mathsf{FV}(\sigma_1)$, so that $\varsigma_2 \circ \sigma_1$ and $\sigma_1 \circ \varsigma_2$ are not necessarily equal.

Proposition 51 *If $\mathsf{supp}(\varsigma) \# \mathsf{FV}(\sigma)$, then $(\varsigma \circ \sigma) = (\sigma \circ \varsigma)$.*

Proof Since ς is a variable allocation, $\mathsf{cosupp}(\varsigma) \# \mathsf{supp}(\sigma)$, by which the result follows. □

Proposition 52 *For all γ_1 and γ_2, $\gamma_1 \circledast \gamma_2$ and $\gamma_1 \circ \gamma_2$ are defined at the same time, and if (γ_1, γ_2) does not interfere, then $(\gamma_1 \circledast \gamma_2) \equiv (\gamma_1 \circ \gamma_2)$.*

Proof By definition of composition and interference. □

5.1.7 Preservation of some reductions inside generalized contexts

In this section, before presenting the top-level translation, we collect two small results about preservation of certain reductions inside certain generalized contexts.

First, we remark that not every reduction is preserved inside generalized contexts, since for instance, rules $\text{LET}_\mathtt{a}$ and $\text{EMPTYLET}_\mathtt{a}$ are only valid at top-level. However, inside generalized bindings, reduction is preserved. Note that every generalized dummy allocation is a generalized binding, so the following result also applies to generalized dummy allocations.

Proposition 53 *For all C_1, C_2, and β, if $C_1 \longrightarrow C_2$, then $\beta[C_1] \longrightarrow \beta[C_2]$.*

Proof By case analysis on the applied rule. □

Moreover, we note that rule $\text{ALLOC}_\mathtt{a}$ is preserved by generalized context application.

Proposition 54 *For all generalized contexts γ, and configurations C and C', if $C \xrightarrow{\text{ALLOC}_\mathtt{a}} C'$, then $\gamma[C] \xrightarrow{\text{ALLOC}_\mathtt{a}} \gamma[C']$.*

Proof Follows from composability of allocation contexts: for all α_1 and α_2, $\alpha_1[\alpha_2]$ is an allocation context. □

5.1.8 The top-level translation

We now present the top-level translation $\lfloor\cdot\rfloor^{\text{TOP}}$, defined in Figure 27, as a function from $\lambda_\circ$ (α-equivalence classes of) expressions to $\lambda_\mathtt{a}$ configurations (it is well defined). As explained above, generalized bindings are used to record the already translated definitions along the translation of top-level bindings, preserving the distinction between variable allocations ς and ordinary substitutions σ. Variable allocations that must alter previous translations are those generated by the translation of a $=_{[n]}$ definition, since only those can be forward referenced.

We first define the top-level translation without checking the validity of the involved generalized context compositions. They are checked shortly afterwards.

The top-level translation handles the size-respecting part of top-level bindings with the function Top. This function expects a size-respecting binding. When its argument is the empty binding, it returns the empty generalized binding. For non-empty bindings, the definitions are translated as sketched above. For a definition of unknown size $x =_{[?]} v$, v is translated by $\lfloor\cdot\rfloor$ to $\mathtt{Rec}\,H\,\mathtt{in}\,V$, and is included in the translation as the generalized binding $\langle H\ ;\ \Box\ ;\ [x \mapsto V]\ ;\ \mathsf{id}\rangle$. A definition of known size $x =_{[n]} v$ is translated into a heap and a variable allocation: v has a translation of the shape $\mathtt{Rec}\,H\,\mathtt{in}\,\ell$, and it is included in the translation of b_v as $\langle H\ ;\ \Box\ ;\ \mathsf{id}\ ;\ [x \mapsto \ell]\rangle$. The top-level translation of an evaluated binding is the composition of the translations of its definitions. If the result is some $\langle H\ ;\ \Box\ ;\ \sigma\ ;\ \varsigma\rangle$, then the variable allocation is applied after the ordinary substitution, which allows the correct treatment of forward references, as sketched in Section 5.1.3.

The two other functions, TDum and TUp, are defined as announced in the beginning of Section 5.1.3. The three functions return generalized contexts: TDum returns a generalized dummy allocation $\langle H\ ;\ \Box\ ;\ \mathsf{id}\ ;\ \varsigma\rangle$, TUp returns $\langle H\ ;\ B\ ;\ \mathsf{id}\ ;\ \mathsf{id}\rangle$, which (by the notation of Section 5.1.3) is $\langle H\ ;\ \mathtt{let}\ B\ \mathtt{in}\ \Box\ ;\ \mathsf{id}\ ;\ \mathsf{id}\rangle$ if $B \neq \varepsilon$, and $\langle H\ ;\ \Box\ ;\ \mathsf{id}\ ;\ \mathsf{id}\rangle$ otherwise.

In case the whole binding (b_v, b) is evaluated (i.e., b is empty), the contexts for pre-allocation and update, $\mathsf{TDum}(b)$ and $\mathsf{TUp}(b)$ are empty, and $\lfloor\mathtt{rec}\ b_\mathsf{v}, b\ \mathtt{in}\ e\rfloor^{\text{TOP}}$ is $\lfloor e\rfloor$,

$$
\begin{array}{lll}
\lfloor \mathtt{rec}\ b_{\mathsf{v}}\ \mathtt{in}\ e \rfloor^{\mathrm{TOP}} & \equiv & \lfloor b_{\mathsf{v}} \rfloor^{\mathrm{TOP}} [\lfloor e \rfloor] \\
\lfloor \mathtt{rec}\ b\ \mathtt{in}\ e \rfloor^{\mathrm{TOP}} & \equiv & \lfloor b \rfloor^{\mathrm{TOP}} [\mathtt{Rec}\,\varepsilon\ \mathtt{in}\ [\![e]\!]] \qquad \text{if } b \text{ is not size-respecting} \\
\lfloor e \rfloor^{\mathrm{TOP}} & \equiv & \lfloor e \rfloor \qquad\qquad \text{if } e \text{ is not of the form } \mathtt{rec}\ b\ \mathtt{in}\ e'
\end{array}
$$

$$
\begin{array}{lll}
\lfloor b_{\mathsf{v}}, b \rfloor^{\mathrm{TOP}} & \equiv & \mathsf{TDum}(b) \circledast \mathsf{Top}(b_{\mathsf{v}}) \circledast \mathsf{TUp}(b) \\
 & & \qquad \text{where } b \text{ does not begin with a} \\
 & & \qquad \text{size-respecting definition.}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{Top}(x =_{[?]} v) & \equiv & \langle H\,;\,\Box\,;\,[x \mapsto V]\,;\,\mathsf{id} \rangle \quad \text{if } \lfloor v \rfloor \equiv \mathtt{Rec}\,H\ \mathtt{in}\ V \\
\mathsf{Top}(x =_{[n]} v) & \equiv & \langle H\,;\,\Box\,;\,\mathsf{id}\,;\,[x \mapsto \ell] \rangle \quad \text{if } \lfloor v \rfloor \equiv \mathtt{Rec}\,H\ \mathtt{in}\ \ell \\
 & & \qquad\qquad\qquad\qquad\quad \text{and } \mathsf{size}(v) = n \\
\mathsf{Top}(\varepsilon) & \equiv & \langle \varepsilon\,;\,\Box\,;\,\mathsf{id}\,;\,\mathsf{id} \rangle \\
\mathsf{Top}(x \diamond v, b_{\mathsf{v}_0}) & \equiv & \mathsf{Top}(x \diamond v) \circledast \mathsf{Top}(b_{\mathsf{v}_0})
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{TDum}(\varepsilon) & \equiv & \langle \varepsilon\,;\,\Box\,;\,\mathsf{id}\,;\,\mathsf{id} \rangle \\
\mathsf{TDum}(x =_{[?]} e, b) & \equiv & \mathsf{TDum}(b) \\
\mathsf{TDum}(x =_{[n]} e, b) & \equiv & (\langle \ell = \mathtt{alloc}\,n\,;\,\Box\,;\,\mathsf{id}\,;\,[x \mapsto \ell] \rangle) \circ \mathsf{TDum}(b)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{TUp}(\varepsilon) & \equiv & \langle \varepsilon\,;\,\Box\,;\,\mathsf{id}\,;\,\mathsf{id} \rangle \\
\mathsf{TUp}(x =_{[?]} e, b) & \equiv & \langle H\,;\,(x = E, \mathsf{Update}(b))\,;\,\mathsf{id}\,;\,\mathsf{id} \rangle \\
 & & \qquad\qquad\qquad\qquad \text{if } \lfloor e \rfloor \equiv \mathtt{Rec}\,H\ \mathtt{in}\ E \\
\mathsf{TUp}(x =_{[n]} e, b) & \equiv & \langle H\,;\,(\_ = (\mathtt{update}\ x\ E), \mathsf{Update}(b))\,;\,\mathsf{id}\,;\,\mathsf{id} \rangle \\
 & & \qquad\qquad\qquad\qquad \text{if } \lfloor e \rfloor \equiv \mathtt{Rec}\,H\ \mathtt{in}\ E
\end{array}
$$

Fig. 27 The top-level translation from $\lambda_\circ$ to λ_{a}

put in the context $\mathsf{Top}(b_{\mathsf{v}})$. Otherwise, $\lfloor \mathtt{rec}\ b_{\mathsf{v}}, b\ \mathtt{in}\ e \rfloor^{\mathrm{TOP}}$ is $\mathtt{Rec}\,\varepsilon\ \mathtt{in}\ [\![e]\!]$, put in the context $\mathsf{TDum}(b) \circledast \mathsf{Top}(b_{\mathsf{v}}) \circledast \mathsf{TUp}(b)$. Notice that there is no context interference, since the innermost one, $\mathsf{TUp}(b)$, does not have any variable allocation, and the outermost one, $\mathsf{TDum}(b)$, has no substitution (but only a variable allocation). So, we could equivalently use $\mathsf{TDum}(b) \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}(b)$. We have two easy results on answers and faulty terms:

Proposition 55 *The function* $\lfloor \cdot \rfloor^{\mathrm{TOP}}$ *maps answers to answers.*

Proof A simple case inspection. □

Proposition 56 *For* e_0 *of one of the shapes in* (3) *of Proposition 7,* $\lfloor e_0 \rfloor^{\mathrm{TOP}}$ *is faulty.*

Proof By case inspection. □

Finally, we prove that all the generalized context compositions we use are well-defined and associative.

Proposition 57 *For all b, x, v, and* b_{v}*, the following hold*

$$
\begin{array}{lll}
b \vdash \mathsf{TDum}(b), & & b \vdash \mathsf{TUp}(b), \\
x \diamond v \vdash \mathsf{Top}(x \diamond v), & \textit{and} & b_{\mathsf{v}} \vdash \mathsf{Top}(b_{\mathsf{v}}).
\end{array}
$$

As a corollary, all the possible generalized contexts resulting from the top-level translation may be composed (by Lemma 43) in an associative fashion (by Lemma 44). This justifies the absence of parentheses in the definition.

5.2 Quotient of $\lambda_{\mathtt{a}}$

In this section, we relate the three translation functions $[\![\cdot]\!]$, $\lfloor\cdot\rfloor$, and $\lfloor\cdot\rfloor^{\text{TOP}}$: we show that their results are equivalent modulo the rules $\text{UPDATE}_{\mathtt{a}}$, $\text{LET}_{\mathtt{a}}$, $\text{EMPTYLET}_{\mathtt{a}}$, $\text{WEAKGC}_{\mathtt{a}}$, and $\text{ALLOC}_{\mathtt{a}}$. So, letting $\overline{\lambda}_{\mathtt{a}}$ be the quotient of $\lambda_{\mathtt{a}}$ modulo these rules, we obtain that they are equal as functions from $\lambda_{\circ}$ to $\overline{\lambda}_{\mathtt{a}}$. Then, we study the compositionality of this function.

Definition 58 ($\overline{\lambda}_{\mathtt{a}}$) Define $=_{\overline{\mathtt{a}}}$ as the smallest equivalence relation over $\lambda_{\mathtt{a}}$ containing the rules $\text{UPDATE}_{\mathtt{a}}$, $\text{LET}_{\mathtt{a}}$, $\text{EMPTYLET}_{\mathtt{a}}$, $\text{WEAKGC}_{\mathtt{a}}$, and $\text{ALLOC}_{\mathtt{a}}$. Let $\overline{\lambda}_{\mathtt{a}}$ be the set of $=_{\overline{\mathtt{a}}}$-equivalence classes. Let reduction in $\overline{\lambda}_{\mathtt{a}}$, written $\longrightarrow_{\overline{\mathtt{a}}}$, be defined by the rules:

$$\frac{C_1 =_{\overline{\mathtt{a}}} C_1' \qquad C_1' \xrightarrow{R} C_2' \qquad C_2' =_{\overline{\mathtt{a}}} C_2}{C_1 \longrightarrow_{\overline{\mathtt{a}}} C_2}$$

where R ranges over the other rules ($\text{BETA}_{\mathtt{a}}$, $\text{PROJECT}_{\mathtt{a}}$, $\text{LIFT}_{\mathtt{a}}$, and $\text{IM}_{\mathtt{a}}$).

Define $=_{\text{ALLOC}_{\mathtt{a}}} \subseteq =_{\overline{\mathtt{a}}}$ to be $\lambda_{\mathtt{a}}$ convertibility by rule $\text{ALLOC}_{\mathtt{a}}$.

We obtain that $\overline{\lambda}_{\mathtt{a}}$ and $\lambda_{\mathtt{a}}$ behave identically:

Lemma 59 *For all C, C reduces to an answer, loops, or is faulty in $\lambda_{\mathtt{a}}$ iff it does in $\overline{\lambda}_{\mathtt{a}}$.*

Proof We show the following:

1. If $C \longrightarrow^* A$, then $C \longrightarrow_{\overline{\mathtt{a}}}^* \mathsf{repr}(A)$. The reduction sequence in $\lambda_{\mathtt{a}}$ is one in $\overline{\lambda}_{\mathtt{a}}$ where some steps become equalities.
2. Conversely, a reduction sequence to an answer in $\overline{\lambda}_{\mathtt{a}}$ corresponds to a sequence of reductions and anti-reductions in $\lambda_{\mathtt{a}}$, which by strong commutation (Lemma 12) lead to a sequence of reductions.
3. If C loops in $\lambda_{\mathtt{a}}$, then C also loops in $\overline{\lambda}_{\mathtt{a}}$, because any infinite reduction sequence involves an infinite number of rules $\text{BETA}_{\mathtt{a}}$ and $\text{PROJECT}_{\mathtt{a}}$.
4. Conversely, we obtain from any infinite reduction sequence in $\overline{\lambda}_{\mathtt{a}}$ an infinite sequence of reductions and anti-reductions in $\lambda_{\mathtt{a}}$, with an infinite number of $\text{BETA}_{\mathtt{a}}$ and $\text{PROJECT}_{\mathtt{a}}$ reductions and no such anti-reduction. By strong commutation, this yields an infinite reduction sequence in $\lambda_{\mathtt{a}}$.
5. Finally, faulty configurations are the same in both calculi.

□

5.2.1 Equating the three translations

We first show that the three translations coincide as functions to $\overline{\lambda}_{\mathtt{a}}$. First, we have the following for the allocating translation.

Proposition 60 *For all v, $\lfloor v\rfloor \equiv \lfloor v\rfloor^{\text{TOP}}$.*

Proof By definition of $\lfloor\cdot\rfloor^{\text{TOP}}$. □

Proposition 61 *For all v, $(\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![v]\!]) =_{\text{ALLOC}_{\mathtt{a}}} \lfloor v\rfloor$.*

Proof Trivial for variables. For other values, apply Proposition 16. □

Proposition 62 *For all e, $(\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e]\!]) =_{\text{ALLOC}_{\mathtt{a}}} \lfloor e\rfloor$.*

Proof By induction on e, using Propositions 61 and 54. □

Consider now the top-level translation of bindings. It splits the bindings in two, cutting at the first non-size-respecting or non-evaluated definition. But of course, one could split at another point, provided the first part is size-respecting. Indeed, the first part is given as an argument to the Top function, which is defined only on size-respecting, 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.

Definition 63 (Partial translation) For all $b \equiv (b_\mathsf{v}, b')$, let the *partial translation of b up to* b_v be $\mathsf{TDum}(b') \circ \mathsf{Top}(b_\mathsf{v}) \circ \mathsf{TUp}(b')$.

The partial translation of b up to b_v is its top-level translation, computed as if b' did not begin with a size-respecting definition. In fact, any partial translation is $=_{\overline{\mathsf{a}}}$-equivalent to the top-level translation, as we now show, using the following properties of the functions TDum and TUp, and of substitution.

Proposition 64 *For all* $C \equiv (\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,E)$*, and* b*,* $\mathsf{Update}(b)[C] =_{\textsc{Alloc}_\mathsf{a}} \mathsf{TUp}(b)[C]$*, using the notation of Section 5.1.3 for coercing bindings to generalized contexts.*

Proof By Propositions 62 and 54. □

Proposition 65 *For all* b, B*, and* E*,*

$$\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\,\mathsf{Dummy}(b), B\,\mathtt{in}\,E =_{\overline{\mathsf{a}}} \mathsf{TDum}(b)[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\,B\,\mathtt{in}\,E].$$

Proof By induction on b and rules $\textsc{Alloc}_\mathsf{a}$ and $\textsc{Let}_\mathsf{a}$. □

Proposition 66 *For all* V, σ*, and* $x \notin \mathsf{FV}(\sigma)$*,* $[x \mapsto \sigma(V)] \circ \sigma = \sigma \circ [x \mapsto V]$.

The key lemma (67) then states that the in-place update machinery indeed computes the expected recursive definition. Hypothesis 17 is crucial here, ensuring that the update is valid.

Lemma 67 *For all* $C \equiv \mathtt{Rec}\,\varepsilon\,\mathtt{in}\,E$ *and size-respecting* $b_{\mathsf{v}_0} \equiv (b_\mathsf{v}, x \diamond v)$*, it holds that*

$$\begin{aligned}&(\mathsf{TDum}(x \diamond v) \circ \mathsf{Top}(b_\mathsf{v}) \circ \mathsf{TUp}(x \diamond v, b))[C]\\ &=_{\overline{\mathsf{a}}} (\mathsf{Top}(b_{\mathsf{v}_0}) \circ \mathsf{Update}(b))[C],\end{aligned}$$

using the notation of Section 5.1.3 for coercing bindings to generalized contexts.

Proof Let $\delta_x \equiv \mathsf{TDum}(x \diamond v) \equiv \langle H_{dx} \,;\, \Box \,;\, \mathsf{id} \,;\, \varsigma_{dx} \rangle$ and $\beta_{b_\mathsf{v}} \equiv \mathsf{Top}(b_\mathsf{v}) \equiv \langle H_{b_\mathsf{v}} \,;\, \Box \,;\, \sigma_{b_\mathsf{v}} \,;\, \varsigma_{b_\mathsf{v}} \rangle$. Let also $C_1 \equiv \delta_x \circ \beta_{b_\mathsf{v}} \circ \mathsf{TUp}(x \diamond v, b)[C]$ and $C_2 \equiv (\mathsf{Top}(b_{\mathsf{v}_0}) \circ \mathsf{Update}(b))[C]$.

First, we have $\mathsf{Top}(b_{\mathsf{v}_0}) \equiv \beta_{b_\mathsf{v}} \circledast \mathsf{Top}(x \diamond v)$.

Then, we proceed by case analysis on $x \diamond v$.

– $(x \diamond v) \equiv (x =_{\lfloor n \rfloor} v)$ with $\mathsf{size}(v) = n$. Then, v is not a variable. Thus, by definition of $\lfloor \cdot \rfloor$, $\lfloor v \rfloor$ has the shape $\mathtt{Rec}\,\ell = S\,\mathtt{in}\,\ell$, for some $\ell \notin \mathsf{FV}(S)$. By α-equivalence, we may choose another fresh location ℓ' such that $\lfloor v \rfloor \equiv (\mathtt{Rec}\,\ell' = S\,\mathtt{in}\,\ell')$. It then holds that $H_{dx} \equiv (\ell = \mathtt{alloc}\ n)$ and $\varsigma_{dx} = [x \mapsto \ell]$, for some ℓ.
Let $\sigma_1 = (\varsigma_{dx} + \varsigma_{b_\mathsf{v}}) \circ \sigma_{b_\mathsf{v}}$. We have:

$$\begin{aligned}C_1 \equiv\ & (\mathtt{Rec}\,\ell = \mathtt{alloc}\,n, \ell' = \sigma_1(S), \sigma_1(H_{b_\mathsf{v}})\\ & \mathtt{in}\ \mathtt{let}\ \_ = \mathtt{update}\ \ell\ \ell', \sigma_1(\mathsf{Update}(b))\ \mathtt{in}\ \sigma_1(E))\\ =_{\overline{\mathsf{a}}}\ & (\mathtt{Rec}\,\ell = \sigma_1(S), \ell' = \sigma_1(S), \sigma_1(H_{b_\mathsf{v}})\\ & \mathtt{in}\ \mathtt{let}\ \sigma_1(\mathsf{Update}(b))\ \mathtt{in}\ \sigma_1(E))\\ & \qquad\qquad (\text{by rules } \textsc{Update}_\mathsf{a} \text{ and } \textsc{Let}_\mathsf{a}),\end{aligned}$$

because $\mathsf{Size}(\sigma_1(S)) = \mathsf{Size}(S) = \mathsf{size}(v) = n = \mathsf{Size}(\mathtt{alloc}\, n)$, by Hypotheses 10 and 17. But then, ℓ' is unused, so the obtained configuration reduces by rule $\textsc{WeakGC}_\mathtt{a}$ to

$$\begin{array}{l} C_1' \equiv \mathtt{Rec}\, \ell = \sigma_1(S), \sigma_1(H_{b_\mathsf{v}})\ \mathtt{in}\ \sigma_1(\mathtt{let}\ \mathsf{Update}(b)\ \mathtt{in}\ E) \\ \equiv (\langle \ell = S, H_{b_\mathsf{v}} \,;\, \Box \,;\, \sigma_{b_\mathsf{v}} \,;\, (\varsigma_{dx} + \varsigma_{b_\mathsf{v}})\rangle)[\mathtt{Rec}\, \varepsilon\ \mathtt{in}\ \mathtt{let}\ \mathsf{Update}(b)\ \mathtt{in}\ E] \\ \equiv (\mathsf{Top}(b_{\mathsf{v}_0}) \circ \mathsf{Update}(b))[C] \equiv C_2. \end{array}$$

– $(x \diamond v) \equiv (x =_{[?]} v)$. Then, $\delta_x \equiv \langle \varepsilon \,;\, \Box \,;\, \mathsf{id} \,;\, \mathsf{id}\rangle$. Let $\lfloor v \rfloor \equiv \mathtt{Rec}\, H_v\ \mathtt{in}\ V$. We have $\mathsf{Top}(x \diamond v) \equiv \langle H_v \,;\, \Box \,;\, [x \mapsto V] \,;\, \mathsf{id}\rangle$.
Now, let $H_1 \equiv H_v, H_{b_\mathsf{v}}$ and $\sigma_1 = \varsigma_{b_\mathsf{v}} \circ \sigma_{b_\mathsf{v}}$. We have $C_1 \equiv \mathtt{Rec}\, \sigma_1(H_1)\ \mathtt{in}\ \sigma_1(\mathtt{let}\ x = V, \mathsf{Update}(b)\ \mathtt{in}\ E)$. By rule $\textsc{Let}_\mathtt{a}$, we have

$$C_1 =_{\overline{\mathtt{a}}} \mathtt{Rec}\, \sigma_1(H_1)\ \mathtt{in}\, [x \mapsto \sigma_1(V)](\sigma_1(\mathtt{let}\ \mathsf{Update}(b)\ \mathtt{in}\ E)).$$

But b_1 may not contain forward references to definitions of unknown size, so the definitions of b_{v_0} can not depend on x. So, $\sigma_1(H_1) \equiv [x \mapsto \sigma_1(V)](\sigma_1(H_1))$, and moreover, by Proposition 66, we have $[x \mapsto \sigma_1(V)] \circ \sigma_1 = \sigma_1 \circ [x \mapsto V]$. So, the obtained configuration is equal to $\mathtt{Rec}\, (\sigma_1 \circ [x \mapsto V])(H_1)\ \mathtt{in}\, (\sigma_1 \circ [x \mapsto V])(\mathtt{let}\ \mathsf{Update}(b)\ \mathtt{in}\ E)$, which is C_2, since $\sigma_1 \circ [x \mapsto V] = \varsigma_{b_\mathsf{v}} \circ (\sigma_{b_\mathsf{v}} \circ [x \mapsto V])$.

□

We then obtain the following.

Lemma 68 *For all $b_\mathsf{v}, b_{\mathsf{v}_0}, b$, and $C \equiv \mathtt{Rec}\, \varepsilon\, \mathtt{in}\, E$, if $(b_\mathsf{v}, b_{\mathsf{v}_0})$ is size-respecting, then* $(\mathsf{TDum}(b_{\mathsf{v}_0}) \circ \mathsf{Top}(b_\mathsf{v}) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b))[C] =_{\overline{\mathtt{a}}} (\mathsf{Top}(b_\mathsf{v}, b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b))[C]$.

Proof By induction on b_{v_0}. The base case is obvious. For the induction step, assume that $b_{\mathsf{v}_0} \equiv (x \diamond v, b_{\mathsf{v}_1})$. We have $\mathsf{TDum}(b_{\mathsf{v}_0}) \equiv \mathsf{TDum}(x \diamond v), \mathsf{TDum}(b_{\mathsf{v}_1})$. By Lemma 67,

$$\begin{array}{l} (\mathsf{TDum}(x \diamond v) \circ \mathsf{Top}(b_\mathsf{v}) \circ \mathsf{TUp}(x \diamond v, b_{\mathsf{v}_1}, b))[C] \\ =_{\overline{\mathtt{a}}} (\mathsf{Top}(b_\mathsf{v}, x \diamond v) \circ \mathsf{Update}(b_{\mathsf{v}_1}, b))[C] \\ =_{\overline{\mathtt{a}}} (\mathsf{Top}(b_\mathsf{v}, x \diamond v) \circ \mathsf{TUp}(b_{\mathsf{v}_1}, b))[C] \quad \text{(by Proposition 64).} \end{array}$$

This obviously gives (using Proposition 49)

$$\begin{array}{l} (\mathsf{TDum}(x \diamond v, b_{\mathsf{v}_1}) \circ \mathsf{Top}(b_\mathsf{v}) \circ \mathsf{TUp}(x \diamond v, b_{\mathsf{v}_1}, b))[C] \\ \equiv \mathsf{TDum}(b_{\mathsf{v}_1})[(\mathsf{TDum}(x \diamond v) \circ \mathsf{Top}(b_\mathsf{v}) \circ \mathsf{TUp}(x \diamond v, b_{\mathsf{v}_1}, b))[C]] \\ =_{\overline{\mathtt{a}}} (\mathsf{TDum}(b_{\mathsf{v}_1}) \circ \mathsf{Top}(b_\mathsf{v}, x \diamond v) \circ \mathsf{TUp}(b_{\mathsf{v}_1}, b))[C]. \end{array}$$

By induction hypothesis, we obtain

$$\begin{array}{l} (\mathsf{TDum}(b_{\mathsf{v}_1}) \circ \mathsf{Top}(b_\mathsf{v}, x \diamond v) \circ \mathsf{TUp}(b_{\mathsf{v}_1}, b))[C] \\ =_{\overline{\mathtt{a}}} (\mathsf{Top}(b_\mathsf{v}, b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b))[C], \end{array}$$

which gives the expected result. □

Lemma 69 *For all b and E,*

$$\mathtt{Rec}\, \varepsilon\ \mathtt{in}\ \mathtt{let}\ \mathsf{Dummy}(b), \mathsf{Update}(b)\ \mathtt{in}\ E =_{\overline{\mathtt{a}}} \lfloor b \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, \varepsilon\ \mathtt{in}\ E].$$

Proof First, if b is empty, then the results holds by application of rule EMPTYLET$_{\mathtt{a}}$, which is included in $=_{\overline{\mathtt{a}}}$.

Otherwise, we have

$$\begin{array}{ll}
& \mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\,\mathsf{Dummy}(b),\mathsf{Update}(b)\,\mathtt{in}\,E \\
=_{\overline{\mathtt{a}}} & \mathsf{TDum}(b)[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\,\mathsf{Update}(b)\,\mathtt{in}\,E] \text{ (By Proposition 65)} \\
\equiv & \mathsf{TDum}(b)[\mathsf{Update}(b)[C]] \qquad \text{(For } C \equiv \mathtt{Rec}\,\varepsilon\,\mathtt{in}\,E\text{)} \\
=_{\overline{\mathtt{a}}} & \mathsf{TDum}(b)[\mathsf{TUp}(b)[C]] \qquad \text{(By Proposition 64)} \\
=_{\overline{\mathtt{a}}} & (\mathsf{TDum}(b) \circ \mathsf{TUp}(b))[C] \qquad \text{(By Proposition 49).}
\end{array}$$

Now, b may be decomposed as $b \equiv (b_{\mathsf{v}_0}, b_0)$, where b_0 does not begin with a size-respecting definition. By Lemma 68 with $b_{\mathsf{v}} = \varepsilon$, we have

$$(\mathsf{TDum}(b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b_0))[C] =_{\overline{\mathtt{a}}} (\mathsf{Top}(b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b_0))[C],$$

which gives

$$\begin{array}{ll}
& \mathsf{TDum}(b) \circ \mathsf{TUp}(b))[C] \\
\equiv & \mathsf{TDum}(b_{\mathsf{v}_0}, b_0) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b_0))[C] \\
\equiv & \mathsf{TDum}(b_0)[(\mathsf{TDum}(b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b_0))[C]] \text{(By Proposition 49)} \\
=_{\overline{\mathtt{a}}} & \mathsf{TDum}(b_0)[(\mathsf{Top}(b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b_0))[C]] \qquad \text{(By Proposition 53)} \\
\equiv & (\mathsf{TDum}(b_0) \circ \mathsf{Top}(b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b_0))[C] \qquad \text{(By Proposition 49)} \\
\equiv & \lfloor b \rfloor^{\mathrm{TOP}}[C] \qquad \text{(By definition of } \lfloor\cdot\rfloor^{\mathrm{TOP}}\text{).}
\end{array}$$

□

Corollary 70 *For all b and e,* $\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![\mathtt{rec}\ b\ \mathtt{in}\ e]\!] =_{\overline{\mathtt{a}}} \lfloor b \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e]\!]]$.

Finally, the following lemma states that the three translations $[\![\cdot]\!]$, $\lfloor\cdot\rfloor$, and $\lfloor\cdot\rfloor^{\mathrm{TOP}}$ are equal as functions from $\lambda_\circ$ to $\overline{\lambda}_{\mathtt{a}}$.

Lemma 71 *For all e, it holds that* $(\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e]\!]) =_{\overline{\mathtt{a}}} \lfloor e \rfloor =_{\overline{\mathtt{a}}} \lfloor e \rfloor^{\mathrm{TOP}}$.

Proof Proposition 62 directly implies $(\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e]\!]) =_{\overline{\mathtt{a}}} \lfloor e \rfloor$.

To prove $\lfloor e \rfloor =_{\overline{\mathtt{a}}} \lfloor e \rfloor^{\mathrm{TOP}}$, we proceed by case analysis on e. If e is not of the shape $\mathtt{rec}\ b\ \mathtt{in}\ e'$, then the result follows by definition of $\lfloor\cdot\rfloor^{\mathrm{TOP}}$. Otherwise, by Corollary 70, we have $\lfloor e \rfloor \equiv \mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![\mathtt{rec}\ b\ \mathtt{in}\ e']\!] =_{\overline{\mathtt{a}}} \lfloor b \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e']\!]]$, so we just have to prove $\lfloor b \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e']\!]] =_{\overline{\mathtt{a}}} \lfloor \mathtt{rec}\ b\ \mathtt{in}\ e' \rfloor^{\mathrm{TOP}}$. If b is not size-respecting, then the result holds by definition of $\lfloor\cdot\rfloor^{\mathrm{TOP}}$. Otherwise, we have $\lfloor \mathtt{rec}\ b\ \mathtt{in}\ e' \rfloor^{\mathrm{TOP}} \equiv \lfloor b \rfloor^{\mathrm{TOP}}[\lfloor e' \rfloor]$. But by Proposition 62, $(\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e']\!]) =_{\mathrm{ALLOC}_{\mathtt{a}}} \lfloor e' \rfloor$, so by Proposition 54 we obtain $\lfloor b \rfloor^{\mathrm{TOP}}[\lfloor e' \rfloor] =_{\mathrm{ALLOC}_{\mathtt{a}}} \lfloor b \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e']\!]]$, which gives the expected result. □

5.2.2 Compositionality

For proving that the evaluation of an expression in $\lambda_\circ$ corresponds to the evaluation of its translation in $\lambda_{\mathtt{a}}$, we seek compositionality properties of our translations. The standard translation is obviously compositional, in the following sense.

Definition 72 (Standard translation of contexts) Define $[\![\mathbb{E}]\!]$ by extension of $[\![\cdot]\!]$ on expressions, with $[\![\Box]\!] \equiv \Box$.

Proposition 73 *For all $\mathbb{E}$ and e,* $[\![\mathbb{E}[e]]\!] \equiv [\![\mathbb{E}]\!][[\![e]\!]]$.

$$
\begin{array}{ll}
\mathsf{Gen}(\mathtt{Rec}\,H\,\mathtt{in}\,\varphi) & \equiv \langle H\,;\,\varphi\,;\,\mathsf{id}\,;\,\mathsf{id}\rangle \\
\lfloor \mathbb{F} \rfloor^{\mathrm{TOP}} & \equiv \mathsf{Gen}(\lfloor \mathbb{F} \rfloor) \\
\lfloor \mathtt{rec}\; b_{\mathsf{v}}\; \mathtt{in}\; \mathbb{F} \rfloor^{\mathrm{TOP}} & \equiv \lfloor b_{\mathsf{v}} \rfloor^{\mathrm{TOP}} \circ \mathsf{Gen}(\lfloor \mathbb{F} \rfloor) \\
\lfloor \mathtt{rec}\; b_{\mathsf{v}}, x \diamond \mathbb{F}, b\; \mathtt{in}\; e \rfloor^{\mathrm{TOP}} & \equiv \\
 & \quad \mathsf{TDum}(x \diamond \mathbb{F}, b) \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}_{[\![e]\!]}(x \diamond \mathbb{F}, b) \\
\mathsf{TUp}_E(x \diamond \mathbb{F}, b) & \equiv \langle H\,;\,\mathtt{let}\; t = \varphi, B\; \mathtt{in}\; E\,;\,\mathsf{id}\,;\,\mathsf{id}\rangle \\
 & \quad \text{if } \mathsf{TUp}(x \diamond \mathbb{F}, b) \equiv \langle H\,;\,(t = \varphi, B)\,;\,\mathsf{id}\,;\,\mathsf{id}\rangle
\end{array}
$$

Fig. 28 Top-level translation of contexts from $\lambda_{\circ}$ to λ_{a}

Proof By trivial induction on $\mathbb{E}$. □

However, we have seen that $[\![\cdot]\!]$ does not lend itself to a simulation argument, so we consider the compositionality of $\lfloor\cdot\rfloor^{\mathrm{TOP}}$. We obtain below in Corollary 77 that for all expressions e and evaluation contexts $\mathbb{E}$, $\lfloor \mathbb{E}[e] \rfloor^{\mathrm{TOP}} =_{\overline{\mathsf{a}}} \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor e \rfloor]$. This is not exactly what one could have hoped for ($\lfloor \mathbb{E}[e] \rfloor^{\mathrm{TOP}} =_{\overline{\mathsf{a}}} \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor e \rfloor^{\mathrm{TOP}}]$) but it will be enough to prove correctness of our translation.

Figure 28 defines $\mathsf{Gen}(\mathtt{Rec}\,H\,\mathtt{in}\,\varphi)$ as the obvious generalized context made of H and φ, using the fact that nested lift contexts are allocation contexts. Then, define $\lfloor\cdot\rfloor$ on nested lift contexts by extension of $\lfloor\cdot\rfloor$ on expressions: we consider $\Box$ not to be a value, and put $\lfloor \Box \rfloor \equiv \mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\Box$. For any $\mathbb{F}$, the translation $\lfloor \mathbb{F} \rfloor^{\mathrm{TOP}}$ has the shape $\mathtt{Rec}\,H\,\mathtt{in}\,\varphi$ for some H and φ. This gives

Proposition 74 *For all* $\mathbb{F}$, $\mathsf{Gen}(\lfloor \mathbb{F} \rfloor)$ *is a generalized evaluation context.*

Proof By induction on $\mathbb{F}$ and case analysis on lift contexts. □

Figure 28 then defines the translation of evaluation contexts. The translation $\mathsf{TDum}(x \diamond \mathbb{F}, b)$ has no hole; $\mathsf{TUp}(x \diamond \mathbb{F}, b)$ has two: one from $\mathbb{F}$, plus one for the body of the returned `let`-binding (present for any $\mathsf{TUp}(b)$). The special notation $\mathsf{TUp}_E(x \diamond \mathbb{F}, b)$ fills the latter with E. We obtain the following immediately.

Proposition 75 *For all* $\mathbb{E}$, $\lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}$ *is a generalized evaluation context whose variable allocation is* id.

Proof By case analysis on $\mathbb{E}$ and Proposition 74, $\lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}$ is a generalized evaluation context. By case analysis, we then prove that its variable allocation is id. If $\mathbb{E}$ is a nested lift context, then by definition of Gen, it is the case. In both other cases, $\lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}$ is defined as the weak composition of more than one generalized evaluation context, which by definition has id as its variable allocation. □

This allows stating the expected compositionality result.

Lemma 76 *For* $\mathbb{E} \equiv \Box$ *and all* e, $\lfloor \mathbb{E}[e] \rfloor^{\mathrm{TOP}} \equiv \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor e \rfloor^{\mathrm{TOP}}]$.
For $\mathbb{E} \not\equiv \Box$ *and all* e, *if* $e \notin \mathsf{values}$, *then* $\lfloor \mathbb{E}[e] \rfloor^{\mathrm{TOP}} \equiv \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor e \rfloor]$.
For all $\mathbb{E}$ *and* v, $\lfloor \mathbb{E}[v] \rfloor^{\mathrm{TOP}} =_{\overline{\mathsf{a}}} \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor v \rfloor]$.

Proof The first point is trivial. The second is obtained for lift contexts first (by a simple case analysis), then for nested lift contexts by straightforward induction, and finally by case analysis on $\mathbb{E}$. As for the last point, if $\Box$ is replaced with a value, it may permit the allocating and top-level translations to perform more administrative reductions, as for instance in contexts of the shape $e\,\Box$. The proof uses Lemma 68. □

Corollary 77 (Weak compositionality) *For all* $\mathbb{E}$ *and* e, $\lfloor \mathbb{E}[e] \rfloor^{\text{TOP}} =_{\overline{\mathtt{a}}} \lfloor \mathbb{E} \rfloor^{\text{TOP}}[\lfloor e \rfloor]$.

Full compositionality does not hold: $\lfloor \mathbb{E}[e] \rfloor^{\text{TOP}}$ is not always $=_{\overline{\mathtt{a}}}$-equivalent to $\lfloor \mathbb{E} \rfloor^{\text{TOP}}[\lfloor e \rfloor^{\text{TOP}}]$. The reason is because $=_{\overline{\mathtt{a}}}$ does not include rule $\text{IM}_\mathtt{a}$, as shown by taking $\mathbb{E} \equiv (\mathtt{rec}\, x =_{[?]} \Box \,\mathtt{in}\, x)$ and $e \equiv (\mathtt{rec}\, y =_{[?]} \{X = z\}, z' =_{[?]} y.X \,\mathtt{in}\, y)$. In this case, $\lfloor \mathbb{E}[e] \rfloor^{\text{TOP}} \equiv (\mathtt{Rec}\, \varepsilon \,\mathtt{in}\, \mathtt{let}\, x = (\mathtt{let}\, y = \{X = z\}, z' = y.X \,\mathtt{in}\, y) \,\mathtt{in}\, x)$, and $\lfloor \mathbb{E} \rfloor^{\text{TOP}}[\lfloor e \rfloor^{\text{TOP}}] \equiv (\mathtt{Rec}\, \ell = \{X = z\} \,\mathtt{in}\, \mathtt{let}\, x = (\mathtt{let}\, z' = \ell.X \,\mathtt{in}\, \ell) \,\mathtt{in}\, x)$. An application of $\text{IM}_\mathtt{a}$ is needed in order to relate them.

Further quotienting $\overline{\lambda}_\mathtt{a}$ by this rule might lead to full compositionality. Moreover, we think that it would preserve the good properties of the translation. In particular, $\lambda_\circ$ reductions by rule $\text{IM}_\circ$ cannot be infinite, so non-termination would remain correctly simulated by the translation. However, this is not needed to complete our correctness proof, so we did not investigate it.

5.3 Quotient of $\lambda_\circ$

We now define $\widetilde{\lambda}_\circ$, based on the following notions of binding scraping $b^*_P(x)$ and context scraping $\mathbb{E}^*(v)$. The intuition is that $\mathbb{E}^*(x)$ does much the same work as iterating the rule $\text{SUBST}_\circ$ until a non-variable value is found. Below, we use it to replace rule $\text{SUBST}_\circ$, in the case where $\mathbb{E}$ is dereferencing. In such cases, if there is no non-variable value for x, then $\mathbb{E}[x]$ is faulty, so we do not have to consider it. Technically, $\mathbb{E}^*(x)$ is then undefined.

Example 78 Let $b_\mathsf{v} \equiv (x =_{[?]} \lambda x'.x', y =_{[?]} x)$ and consider $e \equiv (\mathtt{rec}\, b_\mathsf{v} \,\mathtt{in}\, (y\ \{\}))$. In order to reduce to $e' \equiv (\mathtt{rec}\, b_\mathsf{v} \,\mathtt{in}\, ((\lambda x'.x')\ \{\}))$, e takes two $\text{SUBST}_\circ$ steps. In $\widetilde{\lambda}_\circ$, we will directly replace y with $(\mathtt{rec}\, b_\mathsf{v} \,\mathtt{in}\, \Box)^*(y)$, which is $\lambda x'.x'$, and perform the $\text{BETA}_\circ$ step on-the-fly.

Definition 79 (Binding scraping) For all sets P of variables, bindings b (not necessarily size-respecting), and variables $x \in \mathsf{dom}(b)$, define binding scraping recursively by:

$$\begin{array}{ll} b^*_P(x) \equiv b(x) & \text{if } b(x) \notin \mathsf{vars} \text{ or } b(x) \in \mathsf{vars} \setminus \mathsf{dom}(b) \\ b^*_P(x) \equiv \mathtt{cycle} & \text{if } b(x) \in \mathsf{dom}(b) \cap P \\ b^*_P(x) \equiv b^*_{(\{x\} \cup P)}(b(x)) & \text{if } b(x) \in \mathsf{dom}(b) \setminus P. \end{array}$$

For all such b and x, if $b^*_\emptyset(x) \not\equiv \mathtt{cycle}$, define $b^*(x) \equiv b^*_\emptyset(x)$.

Definition 80 (Context scraping)

Define $\mathbb{E}^*(x) \equiv (\mathsf{Binding}(\mathbb{E}))^*(x)$ if $x \in \mathsf{dom}(\mathsf{Binding}(\mathbb{E}))$

$\mathbb{E}^*(v) \equiv v$ if $v \notin \mathsf{vars}$ or $v \in \mathsf{vars} \setminus \mathsf{dom}(\mathsf{Binding}(\mathbb{E}))$.

Let us now prove elementary properties of binding scraping.

Lemma 81 *Binding scraping is well-defined, i.e., for all* b *and* P, b^*_P *is a total function.*

Proof Let the measure μ be defined from pairs of a binding and a set of variables to natural numbers by $\mu(b,P) = |\mathsf{dom}(b) \setminus P|$, the cardinality of $\mathsf{dom}(b) \setminus P$.

First, we notice that if $\mu(b,P) = 0$, then binding scraping immediately returns, on any variable $x \in \mathsf{dom}(b)$. Indeed, if $b(x) \notin \mathsf{vars}$, it returns $b(x)$. Otherwise, if the variable $b(x)$ is in $\mathsf{dom}(b)$, then it is also in P, so $b^*_P(x) = \mathtt{cycle}$, and if the variable $b(x)$ is not in $\mathsf{dom}(b)$, then $b^*_P(x) = b(x)$.

Then, as the measure decreases by 1 at each recursive call, we conclude that $b^*_P(x)$ is well-defined for any $x \in \mathsf{dom}(b)$. □

Proposition 82 *For all b, P, y, and $x \in \mathsf{dom}(b)$, if $b^*_P(x) \equiv y$, then $y \notin \mathsf{dom}(b)$.*

Proof By induction on the proof of $b^*_P(x) \equiv y$. The base case is when $b(x) \equiv y$ and $y \notin \mathsf{dom}(b)$, which gives immediately the expected result. The induction step is when there exists $z \in \mathsf{dom}(b) \setminus P$ such that $b^*_{\{x\}\cup P}(z) \equiv y$. By induction hypothesis, this gives $y \notin \mathsf{dom}(b)$ as expected. □

We now define $\widetilde{\lambda}_\circ$ as having the same expressions as $\lambda_\circ$, but a different reduction relation, written $\longrightarrow_{\tilde{\circ}}$.

Definition 83 Let the *merging* $\mathbb{E}\langle \mathtt{rec}\ b\ \mathtt{in}\ e\rangle$ of $\mathtt{rec}\ b\ \mathtt{in}\ e$ into the context $\mathbb{E}$ be defined as

- $\mathtt{rec}\ b\ \mathtt{in}\ e$ if $\mathbb{E} \equiv \Box$,
- and otherwise the result of normalizing $\mathbb{E}[\mathtt{rec}\ b\ \mathtt{in}\ e]$ w.r.t. rule CONTEXT$_\circ$/LIFT$_\circ$, plus, if $\mathbb{E}$ had a top-level binding, applying IM$_\circ$ or EM$_\circ$ once.

Note that the capture-avoiding side conditions of Definition 83 are always satisfiable by bound variable renaming.

Then, we define $\longrightarrow_{\tilde{\circ}}$ relatively to $\longrightarrow$ by removing rules BETA$_\circ$, PROJECT$_\circ$, and SUBST$_\circ$, and adding the following three rules:

$$\textsc{Beta}'_\circ \quad \frac{\mathbb{E}^*(v_0) = \lambda y.e}{\mathbb{E}[v_0\ v] \longrightarrow_{\tilde{\circ}} \mathbb{E}\langle \mathtt{rec}\ y =_{[?]} v\ \mathtt{in}\ e\rangle} \qquad \textsc{Project}'_\circ \quad \frac{\mathbb{E}^*(v_0) = \{r\}}{\mathbb{E}[v_0.X] \longrightarrow_{\tilde{\circ}} \mathbb{E}[r(X)]}$$

$$\textsc{Update}'_\circ \quad \frac{{b_\mathsf{v}}^*(y) = v \qquad \mathsf{size}(v) = n}{\mathtt{rec}\ b_\mathsf{v}, x =_{[n]} y, b\ \mathtt{in}\ e \longrightarrow_{\tilde{\circ}} \mathtt{rec}\ b_\mathsf{v}, x =_{[n]} v, b\ \mathtt{in}\ e.}$$

Observe that all $\lambda_\circ$ rules are simulated in $\widetilde{\lambda}_\circ$, except rule SUBST$_\circ$. Indeed, rules BETA$_\circ$ and PROJECT$_\circ$ are special cases of rules BETA$'_\circ$ and PROJECT$'_\circ$. Rule SUBST$_\circ$, albeit not directly simulated, yields a simulation w.r.t. our obserables: evaluation answers, non-termination, and faultiness, as we now show.

Lemma 84 *For all $\mathbb{D}$ and $b_\mathsf{v} \equiv \mathsf{Binding}(\mathbb{D})$, for all $x \in \mathsf{vars}$, $v \notin \mathsf{vars}$, and finite sets of variables P, if $x \notin P$ and $b_\mathsf{v}{}^*_P(x) \equiv v$, then there exists a value v' such that $\mathbb{D}(x) \equiv v'$ and $\mathbb{D}[v'] \longrightarrow^* \mathbb{D}[v]$.*

Proof We proceed by induction on the proof of $b_\mathsf{v}{}^*_P(x) \equiv v$.

- If $b_\mathsf{v}(x) \in \mathsf{vars} \setminus \mathsf{dom}(b)$, then $b_\mathsf{v}{}^*_P(x) \not\equiv v$, contradiction.
- If $b_\mathsf{v}(x) \equiv v$, then, taking $v' \equiv v$, we have $\mathbb{D}(x) \equiv b_\mathsf{v}(x) \equiv v$ and $\mathbb{D}[v] \longrightarrow^* \mathbb{D}[v]$ by reflexivity, as expected.
- If $b_\mathsf{v}(x) \in \mathsf{dom}(b) \cap P$, then $b_\mathsf{v}{}^*_P(x) \equiv \mathtt{cycle}$, contradiction.
- If $b_\mathsf{v}(x) \in \mathsf{dom}(b) \setminus P$, let $y = b_\mathsf{v}(x)$. We know $b_\mathsf{v}{}^*_{\{x\}\cup P}(y) \equiv v$, so $y \notin \{x\} \cup P$. Thus, by induction hypothesis, there exists a v'' such that $\mathbb{D}(y) \equiv v''$ and $\mathbb{D}[v''] \longrightarrow^* \mathbb{D}[v]$. But then, $\mathbb{D}[y] \longrightarrow \mathbb{D}[v''] \longrightarrow^* \mathbb{D}[v]$. So, taking $v' \equiv y$, we obtain $\mathbb{D}(x) \equiv v'$ and $\mathbb{D}[v'] \longrightarrow^* \mathbb{D}[v]$.

□

Lemma 85 *For all $\mathbb{D}, x$, and $v \notin \mathsf{vars}$, if $\mathbb{D}[x] \longrightarrow^+ \mathbb{D}[v]$ then $\mathbb{D}^*(x) \equiv v$.*

Proof By Lemma 84, there exists v' such that $\mathbb{D}(x) \equiv v'$ and $\mathbb{D}[v'] \longrightarrow^* \mathbb{D}[v]$, which immediately gives the expected result. □

Finally:

Lemma 86 *For all e, if e reduces to an answer, loops, or is faulty in $\lambda_\circ$, then so it does in $\widetilde{\lambda}_\circ$.*

Proof The lemma says:

1. if $e \longrightarrow^* a$, then $e \longrightarrow_{\mathfrak{F}}^* a$;
2. if e loops in $\lambda_\circ$, i.e., there exists an infinite reduction sequence starting from e, then e also loops in $\widetilde{\lambda}_\circ$;
3. if e is faulty in $\lambda_\circ$, then it is also faulty in $\widetilde{\lambda}_\circ$.

First, consider a reduction sequence from e to a normal form e_1 in $\lambda_\circ$. We prove by induction on its length that

- if e_1 is an answer, then e reduces to an answer in $\widetilde{\lambda}_\circ$, and
- if e_1 has the shape $\mathbb{D}[v]$, i.e., e is faulty in $\lambda_\circ$, then e is faulty in $\widetilde{\lambda}_\circ$ too.

The base case is trivial. For the induction step, if the first reduction step in the given sequence is not $\textsc{Subst}_\circ$, then it is simulated in $\widetilde{\lambda}_\circ$, so we get the expected result by induction hypothesis. Otherwise, $e \equiv \mathbb{D}[x]$ and the first step has the shape $\mathbb{D}[x] \longrightarrow \mathbb{D}[v]$, with $v \equiv \mathbb{D}(x)$. Consider the maximal subsequence of the given reduction sequence having the shape

$$\mathbb{D}[x] \equiv \mathbb{D}[v_0] \xrightarrow{\textsc{Subst}_\circ} \mathbb{D}[v_1] \xrightarrow{\textsc{Subst}_\circ} \ldots \xrightarrow{\textsc{Subst}_\circ} \mathbb{D}[v_n]$$

with each $v_i \not\equiv v_{i+1}$, i.e., rule $\textsc{Subst}_\circ$ applies each time at $\mathbb{D}$. Thus, $n > 0$, and for $i < n$, v_i is a variable.

Now, if $\mathbb{D}[v_n]$ is an answer, then $\mathbb{D}$ has the shape $\texttt{rec}\ \mathbb{B}_{=_{[m]}}\ \texttt{in}\ v'$ with $\mathsf{size}(v_n) = m$, hence $e \longrightarrow_{\mathfrak{F}} e_1$ by rule $\textsc{Update}'_\circ$.

Otherwise, if $\mathbb{D}[v_n]$ is not an answer, but is actually e_1, i.e., is in normal form, then $\mathbb{D}[x]$ is in normal form in $\widetilde{\lambda}_\circ$ and not an answer, hence faulty in both calculi.

Otherwise, if $\mathbb{D}$ has the shape $\texttt{rec}\ \mathbb{B}_{=_{[n']}}\ \texttt{in}\ e'$ for some n', $\mathbb{B}_{=_{[n']}}$, and e', then $e \longrightarrow_{\mathfrak{F}} \mathbb{D}[v_n]$ by rule $\textsc{Update}'_\circ$ in $\widetilde{\lambda}_\circ$, and we conclude by induction hypothesis.

Otherwise, $\mathbb{D}[v_n]$ further reduces by one of $\textsc{Beta}_\circ$ and $\textsc{Project}_\circ$, and then the corresponding rule ($\textsc{Beta}'_\circ$ or $\textsc{Project}'_\circ$) applies in $\widetilde{\lambda}_\circ$, and we again conclude by induction hypothesis.

Finally, to show that non-termination is preserved, given an infinite reduction sequence in $\lambda_\circ$, build one in $\widetilde{\lambda}_\circ$ by the same algorithm: if the first step is not $\textsc{Subst}_\circ$, then it is simulated directly, otherwise, consider the maximal subsequence of $\textsc{Subst}_\circ$ steps. □

We now quotient $\widetilde{\lambda}_\circ$ by $\textsc{EM}_\circ$ and $\textsc{Update}'_\circ$ to obtain $\overline{\lambda}_\circ$.

Definition 87 ($\overline{\lambda}_\circ$) Define $=_{\overline{\circ}}$ as the smallest equivalence relation over $\widetilde{\lambda}_\circ$ containing the rules $\textsc{EM}_\circ$ and $\textsc{Update}'_\circ$. Let the terms of $\overline{\lambda}_\circ$ be the set of $=_{\overline{\circ}}$-equivalence classes. Let reduction in $\overline{\lambda}_\circ$, written $\longrightarrow_{\overline{\circ}}$, be defined by the rules:

$$\frac{e_1 =_{\overline{\circ}} e_1' \qquad e_1' \xrightarrow{R}_{\overline{\circ}} e_2' \qquad e_2' =_{\overline{\circ}} e_2}{e_1 \longrightarrow_{\overline{\circ}} e_2}$$

where R ranges over the other rules ($\textsc{Lift}_\circ$, $\textsc{Context}_\circ$, $\textsc{IM}_\circ$, $\textsc{Beta}'_\circ$, and $\textsc{Project}'_\circ$).

We now show that $\overline{\lambda}_\circ$ simulates $\widetilde{\lambda}_\circ$, and that $\lfloor\cdot\rfloor^{\text{TOP}}$ remains well-defined as a function from $\overline{\lambda}_\circ$ to $\overline{\lambda}_{\mathtt{a}}$. For this, we prove that rules $\text{EM}_\circ$ and $\text{UPDATE}'_\circ$ preserve $\lfloor\cdot\rfloor^{\text{TOP}}$ (modulo $=_{\overline{\mathtt{a}}}$, which $\overline{\lambda}_{\mathtt{a}}$ is quotiented by) and that infinite $\widetilde{\lambda}_\circ$ reductions can not exclusively contain $\text{EM}_\circ$ or $\text{UPDATE}'_\circ$ reductions. For each of these two rules, we start by defining a measure, and show that each rule makes its measure strictly decrease, and preserves the top-level translation. We start with $\text{EM}_\circ$.

Definition 88 (Number of `rec` *nodes)* $\mathsf{Nbletrec}(e)$ is the number of `rec` nodes in e.

Lemma 89 (External merging) *For all e and e', if $e \xrightarrow{\text{EM}_\circ} e'$, then* $\mathsf{Nbletrec}(e) > \mathsf{Nbletrec}(e')$ *and* $\lfloor e\rfloor^{\text{TOP}} =_{\overline{\mathtt{a}}} \lfloor e'\rfloor^{\text{TOP}}$.

Proof Let $e \equiv \mathtt{rec}\ b_{\mathsf{v}}\ \mathtt{in}\ \mathtt{rec}\ b\ \mathtt{in}\ e_0$
and $e' \equiv \mathtt{rec}\ b_{\mathsf{v}}, b\ \mathtt{in}\ e_0$.

Obviously, $\mathsf{Nbletrec}(e) > \mathsf{Nbletrec}(e')$. Furthermore, we have

$$\lfloor e\rfloor^{\text{TOP}} \equiv \mathsf{Top}(b_{\mathsf{v}})[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\ \mathsf{Dummy}(b), \mathsf{Update}(b)\ \mathtt{in}\,[\![e_0]\!]].$$

If b is empty, after applying rule $\text{EMPTYLET}_{\mathtt{a}}$ (which is in $=_{\overline{\mathtt{a}}}$), the configuration becomes $\mathsf{Top}(b_{\mathsf{v}})[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]$, which is $=_{\overline{\mathtt{a}}}$-equivalent to $\mathsf{Top}(b_{\mathsf{v}})[\lfloor e_0\rfloor] \equiv \lfloor e'\rfloor^{\text{TOP}}$.

Otherwise, using Proposition 53, and after checking that $\mathsf{Top}(b_{\mathsf{v}})$ is a generalized binding, we have

$$\begin{aligned}
&\lfloor e\rfloor^{\text{TOP}} \equiv \mathsf{Top}(b_{\mathsf{v}})[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\ \mathsf{Dummy}(b), \mathsf{Update}(b)\ \mathtt{in}\,[\![e_0]\!]]\\
&=_{\overline{\mathtt{a}}} (\mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TDum}(b))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\ \mathsf{Update}(b)\ \mathtt{in}\,[\![e_0]\!]]\\
&\qquad\qquad \text{(by Propositions 53, 49, and 65)}\\
&=_{\overline{\mathtt{a}}} (\mathsf{TDum}(b) \circ \mathsf{Top}(b_{\mathsf{v}}))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,\mathtt{let}\ \mathsf{Update}(b)\ \mathtt{in}\,[\![e_0]\!]]\\
&\qquad \text{(by Proposition 52, since } \mathsf{dom}(b) \# \mathsf{dom}(b_{\mathsf{v}}) \cup \mathsf{FV}(b_{\mathsf{v}}))\\
&=_{\overline{\mathtt{a}}} (\mathsf{TDum}(b) \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}(b))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]\\
&\qquad\qquad \text{(by Propositions 53, 49, and 64)}.
\end{aligned}$$

But then, let $b \equiv (b_{\mathsf{v}_0}, b')$ with b' not beginning with a size-respecting definition. We have

$$\begin{aligned}
&(\mathsf{TDum}(b) \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}(b))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]\\
&\equiv (\mathsf{TDum}(b_{\mathsf{v}_0}, b') \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b'))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]\\
&\equiv (\mathsf{TDum}(b_{\mathsf{v}_0}) \circ \mathsf{TDum}(b') \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b'))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]\\
&\equiv (\mathsf{TDum}(b') \circ \mathsf{TDum}(b_{\mathsf{v}_0}) \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b'))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]\\
&\qquad\qquad \text{(by Proposition 52)}\\
&\equiv \mathsf{TDum}(b')[(\mathsf{TDum}(b_{\mathsf{v}_0}) \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}(b_{\mathsf{v}_0}, b'))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]]\\
&\qquad\qquad \text{(by Proposition 49)}\\
&=_{\overline{\mathtt{a}}} \mathsf{TDum}(b')[(\mathsf{Top}(b_{\mathsf{v}}, b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b'))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]]\\
&\qquad\qquad \text{(by Lemma 68 and Proposition 53)}\\
&\equiv (\mathsf{TDum}(b') \circ \mathsf{Top}(b_{\mathsf{v}}, b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b'))[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]]\\
&\qquad\qquad \text{(by Proposition 49).}
\end{aligned}$$

But if b' is not empty, then this last configuration is exactly $\lfloor e'\rfloor^{\text{TOP}}$. Otherwise, if b' is empty, then $(\mathsf{TDum}(b') \circ \mathsf{Top}(b_{\mathsf{v}}, b_{\mathsf{v}_0}) \circ \mathsf{TUp}(b')) \equiv \mathsf{Top}(b_{\mathsf{v}}, b_{\mathsf{v}_0})$ is a generalized binding. But by Propositions 62 and 53,

$$\mathsf{Top}(b_{\mathsf{v}}, b_{\mathsf{v}_0})[\mathtt{Rec}\,\varepsilon\,\mathtt{in}\,[\![e_0]\!]] =_{\overline{\mathtt{a}}} \mathsf{Top}(b_{\mathsf{v}}, b_{\mathsf{v}_0})[\lfloor e_0\rfloor] \equiv \lfloor e'\rfloor^{\text{TOP}},$$

which gives the expected result. □

Next, we define a measure that strictly decreases by application of rule UPDATE$'_\circ$.

Definition 90 We define $\mathsf{Length}_\mathsf{v}$ as follows:

$$\mathsf{Length}_\mathsf{v}(\mathtt{rec}\ b_\mathsf{v}, b\ \mathtt{in}\ e) = |\mathsf{dom}(b)| \quad \text{where } b \text{ does not begin with a size-respecting definition.}$$

$$\mathsf{Length}_\mathsf{v}(e) = 0 \quad \text{if } e \text{ does not begin with } \mathtt{rec}.$$

The lemma for rule UPDATE$'_\circ$ requires the following properties of the top-level translation, about how variables can be accessed in the translation of a binding.

Lemma 91 *For all $x, v, H, \sigma, \varsigma$, and b_v, if $\mathsf{Top}(b_\mathsf{v}) \equiv \langle H\ ;\ \Box\ ;\ \sigma\ ;\ \varsigma\rangle$ and ${b_\mathsf{v}}^*(x) \equiv v$, then there exist H_v and V such that $\lfloor v \rfloor \equiv \mathtt{Rec}\, H_v\, \mathtt{in}\, V$, $(\varsigma \circ \sigma)(x) = V$ and $H_v \subseteq H$.*

Proof We prove more generally that for all $x, v, H, \sigma, \varsigma, P$ and b_v, if $\mathsf{Top}(b_\mathsf{v}) \equiv \langle H\ ;\ \Box\ ;\ \sigma\ ;\ \varsigma\rangle$ and ${b_\mathsf{v}}^*_P(x) \equiv v$, then there exist H_v and V such that $\lfloor v \rfloor \equiv \mathtt{Rec}\, H_v\, \mathtt{in}\, V$, $(\varsigma \circ \sigma)(x) = V$ and $H_v \subseteq H$.

We proceed by induction on the proof of ${b_\mathsf{v}}^*_P(x) \equiv v$.

The base case amounts to proving the result with the additional hypothesis that $b_\mathsf{v}(x) \equiv v$. For this, we decompose b_v into $b_{\mathsf{v}_0}, x \diamond v, b_{\mathsf{v}_1}$. By definition of Top, we have $\mathsf{Top}(b_\mathsf{v}) \equiv (\mathsf{Top}(b_{\mathsf{v}_0}) \circledast \mathsf{Top}(x \diamond v) \circledast \mathsf{Top}(b_{\mathsf{v}_1}))$. Let $\mathsf{Top}(b_{\mathsf{v}_0}) \equiv \langle H_0\ ;\ \Box\ ;\ \sigma_0\ ;\ \varsigma_0\rangle$, and $\mathsf{Top}(b_{\mathsf{v}_1}) \equiv \langle H_1\ ;\ \Box\ ;\ \sigma_1\ ;\ \varsigma_1\rangle$.

- If v is a variable y, then $\mathsf{Top}(x \diamond v) \equiv \langle \varepsilon\ ;\ \Box\ ;\ [x \mapsto y]\ ;\ \mathsf{id}\rangle$. Let $H_v \equiv \varepsilon$ and $V \equiv y$. We have $\sigma = (\sigma_0 \circ [x \mapsto y] \circ \sigma_1)$ and $\varsigma = (\varsigma_0 \circ \varsigma_1)$. Furthermore, we know $x \notin \mathsf{dom}(\sigma_1)$, and by Proposition 82, $y \notin \mathsf{dom}(b_\mathsf{v})$. This also gives $y \notin \mathsf{dom}(\varsigma) \cup \mathsf{dom}(\sigma)$, because $(\mathsf{dom}(\varsigma) \cup \mathsf{dom}(\sigma)) \subseteq \mathsf{dom}(b_\mathsf{v})$. Thus $(\varsigma \circ \sigma)(x) = (\varsigma \circ \sigma_0)(y) = y$, as expected.
- If v is not a variable, then $\mathsf{Top}(x \diamond v) \equiv \langle H_v\ ;\ \Box\ ;\ \mathsf{id}\ ;\ [x \mapsto \ell]\rangle$, with $\lfloor v \rfloor \equiv \mathtt{Rec}\, H_v\, \mathtt{in}\, \ell$. Take $V = \ell$. We have $\varsigma = (\varsigma_0 \circ [x \mapsto \ell] \circ \varsigma_1)$ and $\sigma = (\sigma_0 \circ \sigma_1)$. But we know $x \notin \mathsf{dom}(\sigma) \cup \mathsf{dom}(\varsigma_1) \cup \mathsf{dom}(\varsigma_0)$, so $(\varsigma \circ \sigma)(x) = (\varsigma_0 \circ [x \mapsto \ell])(x) = \ell = V$ as expected.

For the induction step, assume $b_\mathsf{v}(x) \equiv y$ and ${b_\mathsf{v}}^*_{\{x\} \cup P}(y) \equiv v$. Then, b_v has the shape $(b_{\mathsf{v}_0}, x \diamond y, b_{\mathsf{v}_1})$ for some $b_{\mathsf{v}_0}, b_{\mathsf{v}_1}$. By definition of Top, we have $\mathsf{Top}(b_\mathsf{v}) \equiv (\mathsf{Top}(b_{\mathsf{v}_0}) \circledast \mathsf{Top}(x \diamond y) \circledast \mathsf{Top}(b_{\mathsf{v}_1}))$. Let $\mathsf{Top}(b_{\mathsf{v}_0}) \equiv \langle H_0\ ;\ \Box\ ;\ \sigma_0\ ;\ \varsigma_0\rangle$, and $\mathsf{Top}(b_{\mathsf{v}_1}) \equiv \langle H_1\ ;\ \Box\ ;\ \sigma_1\ ;\ \varsigma_1\rangle$. We know that $H \equiv (H_0, H_1)$, $\sigma = (\sigma_0 \circ [x \mapsto y] \circ \sigma_1)$ and $\varsigma = (\varsigma_0 \circ \varsigma_1)$. By induction hypothesis, there exist H_v and V such that $\lfloor v \rfloor \equiv \mathtt{Rec}\, H_v\, \mathtt{in}\, V$, $H_v \subseteq H$, and $(\varsigma \circ \sigma)(y) = V$. Thus, there only remains to prove that $(\varsigma \circ \sigma)(y) = (\varsigma \circ \sigma)(x)$.

- If $y \in \mathsf{dom}(b_{\mathsf{v}_1})$, then, since b_v contains a forward reference from x to y, y has a known size indication in b_v. So, $y \in \mathsf{dom}(\varsigma_1)$, hence $y \notin \mathsf{dom}(\sigma)$. Thus, $(\varsigma \circ \sigma)(y) = \varsigma(y) = (\varsigma \circ \sigma_0 \circ [x \mapsto y])(x) = (\varsigma \circ \sigma)(x)$.
- If $y \notin \mathsf{dom}(b_{\mathsf{v}_1})$, then $(\varsigma \circ \sigma)(y) = (\varsigma \circ \sigma_0)(y) = (\varsigma \circ \sigma_0 \circ [x \mapsto y])(x) = (\varsigma \circ \sigma)(x)$.

□

Lemma 92 *For all $\mathbb{E}, x, H, \xi, \sigma$, and $v \notin \mathsf{vars}$, if $\lfloor \mathbb{E} \rfloor^{\mathrm{TOP}} \equiv \langle H\ ;\ \xi\ ;\ \sigma\ ;\ \mathsf{id}\rangle$, and $\mathbb{E}^*(x) \equiv v$, then there exist H_v and ℓ such that $\lfloor v \rfloor \equiv \mathtt{Rec}\, H_v\, \mathtt{in}\, \ell$, $\sigma(x) = \ell$ and $H_v \subseteq H$.*

Proof By case analysis on $\mathbb{E}$. First, if $\mathbb{E}$ is a nested lift context, then $v \equiv x$ and $x \notin \mathsf{dom}(\sigma)$, which gives the expected result.

If $\mathbb{E} \equiv (\mathtt{rec}\ b_\mathsf{v}\ \mathtt{in}\ \mathbb{F})$, then $\lfloor \mathbb{E} \rfloor^{\mathrm{TOP}} \equiv \mathsf{Top}(b_\mathsf{v}) \circ \mathsf{Gen}(\lfloor \mathbb{F} \rfloor)$. But by definition, $\mathsf{Gen}(\lfloor \mathbb{F} \rfloor) \equiv \langle H'\ ;\ \varphi\ ;\ \mathsf{id}\ ;\ \mathsf{id}\rangle$ for some H' and φ. So, $\sigma = \mathsf{Subst}(\mathsf{Top}(b_\mathsf{v}))$, and we conclude by Lemma 91.

If $\mathbb{E} \equiv (\texttt{rec}\ b_{\mathsf{v}}, y \diamond \mathbb{F}, b\ \texttt{in}\ e)$, then $\lfloor \mathbb{E} \rfloor^{\text{TOP}} \equiv (\mathsf{TDum}(y \diamond \mathbb{F}, b) \circ \mathsf{Top}(b_{\mathsf{v}}) \circ \mathsf{TUp}_{[\![e]\!]}(y \diamond \mathbb{F}, b))$. Let $\mathsf{TDum}(y \diamond \mathbb{F}, b) \equiv \langle H_0 \,;\, \Box \,;\, \mathsf{id} \,;\, \varsigma_0 \rangle$, $\mathsf{Top}(b_{\mathsf{v}}) \equiv \langle H_{b_{\mathsf{v}}} \,;\, \Box \,;\, \sigma_{b_{\mathsf{v}}} \,;\, \varsigma_{b_{\mathsf{v}}} \rangle$, and $\mathsf{TUp}_{[\![e]\!]}(y \diamond \mathbb{F}, b) \equiv \langle H_1 \,;\, \xi \,;\, \mathsf{id} \,;\, \mathsf{id} \rangle$ (they have these shapes by definition). We have $\sigma = (\varsigma_0 \circ \varsigma_{b_{\mathsf{v}}} \circ \sigma_{b_{\mathsf{v}}})$, $H \equiv (H_0, H_{b_{\mathsf{v}}}, H_1)$. By Lemma 91, we obtain H_v and ℓ (because $v \notin \mathsf{vars}$) such that $\lfloor v \rfloor \equiv \texttt{Rec}\, H_v\ \texttt{in}\ \ell$, $H_v \subseteq H_{b_{\mathsf{v}}}$, and $(\varsigma_{b_{\mathsf{v}}} \circ \sigma_{b_{\mathsf{v}}})(x) = \ell$. Here, this immediately gives $H_v \subseteq H_{b_{\mathsf{v}}} \subseteq H$ and $\sigma(x) = \ell$. □

Lemma 93 *For all e and e', if $e \xrightarrow{\text{UPDATE}'_\circ} e'$, then $\mathsf{Length}_{\mathsf{v}}(e) > \mathsf{Length}_{\mathsf{v}}(e')$ and $\lfloor e \rfloor^{\text{TOP}} =_{\overline{\mathtt{a}}} \lfloor e' \rfloor^{\text{TOP}}$.*

Proof Obviously, $\mathsf{Length}_{\mathsf{v}}(e) > \mathsf{Length}_{\mathsf{v}}(e')$. Furthermore, we have $e \equiv (\texttt{rec}\,(b_{\mathsf{v}}, y =_{[n]} x, b)\ \texttt{in}\ e_0)$. Let $\mathbb{D} \equiv (\texttt{rec}\,(b_{\mathsf{v}}, y =_{[n]} \Box, b)\ \texttt{in}\ e_0)$. Since $\mathbb{D}[x] \xrightarrow{\text{UPDATE}'_\circ} e'$, we have some non-variable value v such that $b_{\mathsf{v}}^*(x) \equiv v$ and $\mathsf{size}(v) = n$. By Lemma 92, and letting $\lfloor \mathbb{D} \rfloor^{\text{TOP}} \equiv \langle H \,;\, \xi \,;\, \sigma \,;\, \mathsf{id} \rangle$, we have $\lfloor v \rfloor \equiv \texttt{Rec}\, H_v\ \texttt{in}\ \ell$ such that $H_v \subseteq H$ and $\sigma(x) = \ell$. Then, let $\delta_b \equiv \mathsf{TDum}(b)$ and $\delta_y \equiv \langle \ell' = \texttt{alloc}\ n \,;\, \Box \,;\, \varsigma_y \,;\, \mathsf{id} \rangle \equiv \mathsf{TDum}(y =_{[n]} \Box)$ for some location ℓ', with $\varsigma_y = [y \mapsto \ell']$. Let $\delta \equiv \delta_y \circ \delta_b$, and $\mathsf{Top}(b_{\mathsf{v}}) \equiv \langle H_1 \,;\, \Box \,;\, \sigma_1 \,;\, \mathsf{id} \rangle$, such that $H_v \subseteq H_1 \subseteq H$ and $\sigma = \mathsf{Subst}(\delta) \circ \sigma_1 = \mathsf{Subst}(\delta_b) \circ \varsigma_y \circ \sigma_1$.

We have

$$\lfloor \mathbb{D}[x] \rfloor^{\text{TOP}} \equiv (\delta \circ \mathsf{Top}(b_{\mathsf{v}}))[\texttt{let \_ = update}\ y\ x, \mathsf{Update}(b)\ \texttt{in}\ [\![e_0]\!]].$$

But $y \notin \mathsf{supp}(\sigma_1)$, so

$$\lfloor \mathbb{D}[x] \rfloor^{\text{TOP}} \equiv (\delta \circ \mathsf{Top}(b_{\mathsf{v}}))[\texttt{let \_ = update}\ \ell'\ \ell, \mathsf{Update}(b)\ \texttt{in}\ [\![e_0]\!]].$$

Furthermore, $\mathsf{size}(v) = \mathsf{Size}(H_v(\ell)) = n$, by Hypothesis 17, and moreover by construction of the translation, H_v only contains one binding. So, in fact, the update copies $\lfloor v \rfloor$ entirely, and the previous configuration reduces by $\text{UPDATE}_{\mathtt{a}}$ and $\text{LET}_{\mathtt{a}}$ to

$$(\delta_b \circ \mathsf{Top}(b_{\mathsf{v}}, y =_{[n]} v))[\texttt{let}\ \mathsf{Update}(b)\ \texttt{in}\ [\![e_0]\!]].$$

Finally, by Proposition 64 and Lemma 68, this is $=_{\overline{\mathtt{a}}}$-equivalent to $\lfloor \mathbb{D}[v] \rfloor^{\text{TOP}}$, which is exactly $\lfloor e' \rfloor^{\text{TOP}}$.

□

We obtain that $\overline{\lambda}_\circ$ simulates $\widetilde{\lambda}_\circ$, and hence also simulates $\lambda_\circ$.

Lemma 94 *For all e, if e reduces to an answer, loops, or is faulty in $\widetilde{\lambda}_\circ$, then so it does in $\overline{\lambda}_\circ$.*

Proof The only non-trivial point is non termination. By Lemmas 89 and 93, and since $\text{UPDATE}'_\circ$ preserves $\mathsf{Nbletrec}$, the lexicographic order $(\mathsf{Nbletrec}, \mathsf{Length}_{\mathsf{v}})$ stricly decreases by $\text{EM}_\circ$ and $\text{UPDATE}'_\circ$. Thus, there is no infinite reduction sequence in $\widetilde{\lambda}_\circ$ involving only these rules. □

5.4 Correctness

We now have the tools to prove the expected correctness theorem. We first notice the following useful property of $\mathbb{E}\langle \mathtt{rec}\ b_{\mathsf{v}}\ \mathtt{in}\ e\rangle$.

Proposition 95 *For all $\mathbb{E}, b_{\mathsf{v}}, e$, if $\mathbb{E}\langle \mathtt{rec}\ b_{\mathsf{v}}\ \mathtt{in}\ e\rangle$ is defined, then $\lfloor \mathbb{E}\langle \mathtt{rec}\ b_{\mathsf{v}}\ \mathtt{in}\ e\rangle \rfloor^{\mathrm{TOP}} \equiv \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}} \circ \lfloor b_{\mathsf{v}} \rfloor^{\mathrm{TOP}}[\lfloor e \rfloor]$.*

Proof By commutation of $\mathsf{Top}(b_{\mathsf{v}})$ with $\mathsf{Gen}(\lfloor \mathbb{F} \rfloor)$, where $\mathbb{F}$ is the nested lift context part of $\mathbb{E}$. □

Lemma 96 (Correctness) *For all e and e', if $e \longrightarrow_{\overline{\circ}} e'$, then $\lfloor e \rfloor^{\mathrm{TOP}} \longrightarrow_{\overline{\mathtt{a}}}^{+} \lfloor e' \rfloor^{\mathrm{TOP}}$.*

Proof We proceed by case analysis on the rule used.

BETA$'_{\circ}$ There exist $\mathbb{E}, v_0$, and v such that $e \equiv \mathbb{E}[v_0\ v]$, $\mathbb{E}^*(v_0) \equiv \lambda x.g$, and $e' \equiv \mathbb{E}\langle \mathtt{rec}\ x \mathtt{=}_{[?]} v\ \mathtt{in}\ g\rangle$. Let $\lfloor v \rfloor \equiv \mathtt{Rec}\, H_v\, \mathtt{in}\, V$ and $\lfloor \lambda x.g \rfloor \equiv \mathtt{Rec}\, H_1\, \mathtt{in}\, \ell$ (with $H_1 \equiv \ell \mathtt{=} \lambda x.[\![g]\!]$). Let $\lfloor \mathbb{E} \rfloor^{\mathrm{TOP}} \equiv \langle H\,;\,\xi\,;\,\sigma\,;\,\mathsf{id}\rangle$.

– If $v_0 \equiv \lambda x.g$, then

$$\begin{array}{rl}
\lfloor e \rfloor^{\mathrm{TOP}} & =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor v_0\ v \rfloor] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_1, H_v\, \mathtt{in}\, \ell\ V] \\
& \longrightarrow_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_1, H_v\, \mathtt{in}\, [x \mapsto V](\![g]\!])] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_v\, \mathtt{in}\, [x \mapsto V](\![g]\!])] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}} \circ \lfloor x \mathtt{=}_{[?]} v \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, \varepsilon\, \mathtt{in}\, [\![g]\!]] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}} \circ \lfloor x \mathtt{=}_{[?]} v \rfloor^{\mathrm{TOP}}[\lfloor g \rfloor] \\
& =_{\overline{\mathtt{a}}}\ \lfloor e' \rfloor^{\mathrm{TOP}}.
\end{array}$$

– If $v_0 \equiv y$ with $\mathbb{E}^*(y) \equiv \lambda x.g$, then by Lemma 92 we have a location $\ell = \sigma(y)$ such that $H(\ell) \equiv \lambda x.[\![g]\!]$. So we have

$$\begin{array}{rl}
\lfloor e \rfloor^{\mathrm{TOP}} & =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor y\ v \rfloor] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_v\, \mathtt{in}\, y\ V] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_v\, \mathtt{in}\, \ell\ V] \\
& \longrightarrow_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_v\, \mathtt{in}\, [x \mapsto V](\![g]\!])] \\
& =_{\overline{\mathtt{a}}}\ \lfloor e' \rfloor^{\mathrm{TOP}} \text{ (as above).}
\end{array}$$

PROJECT$'_{\circ}$ There exist $\mathbb{E}, v_0$, and X such that $e \equiv \mathbb{E}[v_0.X]$, and $\mathbb{E}^*(v_0) \equiv \{r\}$ with $r(X) \equiv z$. Let $\lfloor \{r\} \rfloor \equiv \mathtt{Rec}\, H_1\, \mathtt{in}\, \ell$ (with $H_1 \equiv \ell \mathtt{=} \{r\}$). The whole expression reduces to $\mathbb{E}[z]$. Let $\lfloor \mathbb{E} \rfloor^{\mathrm{TOP}} \equiv \langle H\,;\,\xi\,;\,\sigma\,;\,\mathsf{id}\rangle$.

– If $v_0 \equiv \{r\}$, then

$$\begin{array}{rl}
\lfloor e \rfloor^{\mathrm{TOP}} & =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor v_0.X \rfloor] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_1\, \mathtt{in}\, \ell.X] \\
& \longrightarrow_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, H_1\, \mathtt{in}\, z] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, \varepsilon\, \mathtt{in}\, z] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E}[z] \rfloor^{\mathrm{TOP}}.
\end{array}$$

– If $v_0 \equiv y$ with $\mathbb{E}^*(y) \equiv \{r\}$, then by Lemma 92 we have a location $\ell = \sigma(y)$ such that $H(\ell) \equiv \{r\}$. So we have

$$\begin{array}{rl}
\lfloor e \rfloor^{\mathrm{TOP}} & =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\lfloor y.X \rfloor] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, \varepsilon\, \mathtt{in}\, \ell.X] \\
& \longrightarrow_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E} \rfloor^{\mathrm{TOP}}[\mathtt{Rec}\, \varepsilon\, \mathtt{in}\, z] \\
& =_{\overline{\mathtt{a}}}\ \lfloor \mathbb{E}[z] \rfloor^{\mathrm{TOP}}.
\end{array}$$

CONTEXT$_\circ$ with LIFT$_\circ$ We have $e \equiv \mathbb{E}[e_1]$, with $e_1 \equiv \mathbb{L}[\texttt{rec}\ b\ \texttt{in}\ e_0]$, and $e' \equiv \mathbb{E}[e_2]$, with $e_2 \equiv \texttt{rec}\ b\ \texttt{in}\ \mathbb{L}[e_0]$. Let $\gamma \equiv \langle H\ ;\ \xi\ ;\ \sigma\ ;\ \mathsf{id}\rangle \equiv \lfloor\mathbb{E}\rfloor^{\text{TOP}}$ and $\lfloor\mathbb{L}\rfloor \equiv \langle H_\mathbb{L}\ ;\ \eta_\mathbb{L}\ ;\ \mathsf{id}\ ;\ \mathsf{id}\rangle$. We have

$$\begin{array}{rl} \lfloor\mathbb{E}[e_1]\rfloor^{\text{TOP}} & =_{\overline{\mathtt{a}}}\ \gamma[\texttt{Rec}\,H_\mathbb{L}\ \texttt{in}\ \eta_\mathbb{L}[\texttt{let}\ \mathsf{Dummy}(b), \mathsf{Update}(b)\ \texttt{in}\ [\![e_0]\!]]] \\ & \longrightarrow_{\overline{\mathtt{a}}}\ \gamma[\texttt{Rec}\,H_\mathbb{L}\ \texttt{in}\ \texttt{let}\ \mathsf{Dummy}(b), \mathsf{Update}(b)\ \texttt{in}\ \eta_\mathbb{L}[[\![e_0]\!]]] \\ & =_{\overline{\mathtt{a}}}\ \gamma[\texttt{Rec}\,\varepsilon\ \texttt{in}\ [\![e_2]\!]] \\ & =_{\overline{\mathtt{a}}}\ [\![e']\!]. \end{array}$$

IM$_\circ$ We have $e \equiv \texttt{rec}\ b\ \texttt{in}\ e_0$ and $e' \equiv \texttt{rec}\ b'\ \texttt{in}\ e_0$, with $b \equiv (b_\mathsf{v}, x \diamond (\texttt{rec}\ b_1\ \texttt{in}\ e_1), b_2)$, and $b' \equiv (b_\mathsf{v}, b_1, x \diamond e_1, b_2)$.
Let $\beta_{b_\mathsf{v}} \equiv \mathsf{Top}(b_\mathsf{v})$, $b_0 \equiv (x \diamond (\texttt{rec}\ b_1\ \texttt{in}\ e_1), b_2)$ and $b'_0 \equiv (x \diamond e_1, b_2)$.
By definition of the translation, we have $\lfloor\texttt{rec}\ b\ \texttt{in}\ e_0\rfloor^{\text{TOP}} \equiv \mathsf{TDum}(b_0) \circ \beta_{b_\mathsf{v}} \circ \mathsf{TUp}(b_0)[\texttt{Rec}\,\varepsilon\ \texttt{in}\ [\![e_0]\!]]$.
Let now $(t, \varphi) \equiv \begin{cases} (x, \Box) \text{ if } \diamond = =_{[?]} \\ (\_, \texttt{update}\ x\ \Box) \text{ otherwise.} \end{cases}$
We have

$$\begin{array}{rl} \lfloor\texttt{rec}\ b_1\ \texttt{in}\ e_1\rfloor & \equiv (\texttt{Rec}\,\varepsilon\ \texttt{in}\ [\![\texttt{rec}\ b_1\ \texttt{in}\ e_1]\!]) \\ & \equiv (\texttt{Rec}\,\varepsilon\ \texttt{in}\ \texttt{let}\ \mathsf{Dummy}(b_1), \mathsf{Update}(b_1)\ \texttt{in}\ [\![e_1]\!]) \\ & \equiv \texttt{Rec}\,\varepsilon\ \texttt{in}\ E_1. \end{array}$$

So $\lfloor\texttt{rec}\ b\ \texttt{in}\ e_0\rfloor^{\text{TOP}} \equiv \mathsf{TDum}(b_0) \circ \beta_{b_\mathsf{v}}[\texttt{Rec}\,\varepsilon\ \texttt{in}\ \texttt{let}\ t = \varphi[E_1], \mathsf{Update}(b_2)\ \texttt{in}\ [\![e_0]\!]]$.
But this reduces by (maybe rule LIFT$_\mathtt{a}$ and) rule IM$_\mathtt{a}$ to

$$\begin{array}{l} \mathsf{TDum}(b_0) \circ \beta_{b_\mathsf{v}}[\texttt{Rec}\,\varepsilon\ \texttt{in}\ \texttt{let}\ \mathsf{Dummy}(b_1), \mathsf{Update}(b_1), \\ \qquad\qquad\qquad\qquad t = \varphi[[\![e_1]\!]], \mathsf{Update}(b_2) \\ \qquad\qquad\qquad\quad \texttt{in}\ [\![e_0]\!]]. \end{array}$$

But we recognize $\mathsf{Update}(b_1, b'_0)$, so the obtained configuration is equal to

$$C \equiv \mathsf{TDum}(b_0) \circ \beta_{b_\mathsf{v}}[\texttt{Rec}\,\varepsilon\ \texttt{in}\ \texttt{let}\ \mathsf{Dummy}(b_1), \mathsf{Update}(b_1, b'_0)\ \texttt{in}\ [\![e_0]\!]].$$

Then, by Propositions 53 and 65, we obtain

$$C =_{\overline{\mathtt{a}}} \mathsf{TDum}(b_0) \circ \beta_{b_\mathsf{v}} \circ \mathsf{TDum}(b_1)[\texttt{Rec}\,\varepsilon\ \texttt{in}\ \texttt{let}\ \mathsf{Update}(b_1, b'_0)\ \texttt{in}\ [\![e_0]\!]].$$

But as $\mathsf{dom}(b_1) \# \mathsf{dom}(b_\mathsf{v}) \cup \mathsf{FV}(b_\mathsf{v})$, this is equal to

$$\mathsf{TDum}(b_1, b'_0) \circ \beta_{b_\mathsf{v}}[\texttt{Rec}\,\varepsilon\ \texttt{in}\ \texttt{let}\ \mathsf{Update}(b_1, b'_0)\ \texttt{in}\ [\![e_0]\!]],$$

which by Proposition 64 and Lemma 68 is $=_{\overline{\mathtt{a}}}$-equivalent to $\lfloor\texttt{rec}\ b'\ \texttt{in}\ e_0\rfloor^{\text{TOP}}$, which concludes the proof.

□

This yields:

Corollary 97 *For all e, if e reduces to an answer, loops, or is faulty in $\overline{\lambda}_\circ$, then so does $\lfloor e\rfloor^{\text{TOP}}$ in $\overline{\lambda}_\mathtt{a}$.*

Proof Since $\lfloor\cdot\rfloor^{\mathrm{TOP}}$ maps answers to answers (Proposition 55), if e reduces to an answer, then so does $\lfloor e\rfloor^{\mathrm{TOP}}$. Moreover, because Lemma 96 uses $\longrightarrow_{\overline{\mathtt{a}}}^{+}$, if e loops, so does $\lfloor e\rfloor^{\mathrm{TOP}}$. Finally, if e is faulty, then it reduces to a term e_0 in normal form of one of the shapes in Proposition 55, hence $\lfloor e\rfloor^{\mathrm{TOP}}$ reduces to $\lfloor e_0\rfloor^{\mathrm{TOP}}$, which is faulty by Proposition 56. Hence $\lfloor e\rfloor^{\mathrm{TOP}}$ is faulty. □

We finally have:

Proof (of Theorem 18) By composing Lemmas 86, 94, Corollary 97, and Lemma 59, we obtain the result for $\lfloor e\rfloor^{\mathrm{TOP}}$. But $\lfloor e\rfloor^{\mathrm{TOP}} =_{\overline{\mathtt{a}}} [\![e]\!]$ in $\overline{\lambda}_{\mathtt{a}}$, hence they behave the same by Lemma 59. □

6 Related work

Ariola and Blom [2] study λ-calculi with `let rec`, in relation with the graphs they represent. The $\lambda_\circ$ language presented here is mostly a deterministic variant of their call-by-value calculus. The main difference lies in our size indications, which specialize the language for efficient compilation.

Lang et al [19] study λ-calculi with sharing and recursion, resulting in the notion of *Addressed Term Rewriting Systems*. Unlike in $\lambda_\circ$, cyclic data structures are represented using addresses: each node of a term is given an address, which can be referred to by a *back pointer*. Addresses can be shared among instances of the same term. Moreover, addresses are not bound in the considered term, whereas in $\lambda_\circ$, `rec` does bind variables. Thus, addressed terms must satisfy a number of coherence conditions, which appear to be far from trivial. This explains our choice of Ariola et al.'s approach.

Erkök and Launchbury [10] consider the interaction of recursion with side effects. In the setting of monadic meta-languages, Moggi and Sabry [23] devise an operator named `Mfix`, with an operational semantics, which unifies different language constructs for recursion. This very interesting work is more abstract than ours, in the sense that it unifies several recursion constructs from both eager and lazy languages, whereas our work is specific to call-by-value. Also, we are not specifically interested in the interaction between recursion and side effects, although we treat it with care. Moreover, Erkök and Launchbury and Moggi and Sabry are not concerned with compilation.

Another work on recursion, already discussed in Section 1.2, is Boudol's calculus [3]. From the standpoint of expressive power, this calculus is incomparable with $\lambda_\circ$. On the one hand, the semantics of $\lambda_\circ$, based on Ariola et al.'s work, allows to represent cyclic data structures such as `let rec` x `=` `cons` 1 x, while such a definition loops in Boudol's calculus. On the other hand, the unrestricted `let rec` of Boudol's calculus avoids the difficult guess of correct size indications.

From the standpoint of compilation, Boudol and Zimmer [4] use a backpatching approach, thus increasing the number of run-time tests and indirections. A similar backpatching approach is used in Russo's extension of ML with recursive modules [27], implemented in Moscow ML, and in Dreyer's work on typing of extended recursion [8].

Syme [30] extends the F# language with generalized recursive definitions where the right-hand sides are arbitrary computations. Haskell-style lazy evaluation is used to evaluate these recursive definitions: a strong, forward reference to a recursively-defined variable x is not an error, but causes the definition of x to be evaluated at this point. In the application scenario considered by Syme, namely interfacing with libraries written in object-oriented

languages, no compile-time information is available on dependencies and object sizes, rendering our approach inapplicable and essentially forcing the use of lazy evaluation. However, lazy evaluation has some additional run-time costs and makes evaluation order hard to guess in advance.

Nordlander, Carlsson and Gill [24] describe an original variant of the in-place update scheme where the sizes of the recursively-defined values need not be known at compile-time. Consider a recursive definition $\texttt{rec}x = e$. The variable x is first bound to a unique marker; then, e is evaluated to a value v; finally, the memory representation of v is recursively traversed, replacing all occurrences of the unique marker with a pointer to v. This recursive traversal can be much more costly than the updating of dummy blocks performed by the in-place update scheme: a naive implementation runs in time $O(N)$ where N is the size of the value v. (This size can be arbitrarily large even if the evaluation of e is trivial: consider $\texttt{rec}x = \texttt{Cons}\ l\ x$ where l is a 10^6 element list previously computed.) Assuming linear allocation and a copying garbage collector, the traversal can be restricted to blocks allocated during the evaluation of e, resulting in a reasonable complexity $O(\min(N,M))$ where M is the number of allocations performed by the evaluation of e. However, this improvement seems impossible for memory managers that perform non-linear allocation like those of OCaml and F#.

Mutually-recursive definitions of functions (syntactic λ-abstractions) is a frequently-occurring special case that admits a very efficient implementation [18, 1]. Instead of allocating one closure block for each function, containing pointers to the other closure blocks, it is possible to share a single memory block between the closures, and use pointer arithmetic to recover pointers to the other closures from any given closure. No in-place update is needed to build loops between the closures. We believe that this trick could be combined with a more general in-place update scheme to efficiently compile recursive definitions that combine syntactic λ-abstractions and more general computations. However, significant extensions to λ_{a} would be needed to account for this approach.

7 Conclusions and future work

In this article, we have developed the first formal semantic account of the in-place update scheme, and proved its ability to implement faithfully an extended call-by-value recursion construct, as characterized by our source language $\lambda_{\circ}$.

At this point, one may wonder whether $\lambda_{\circ}$ embodies the most powerful call-by-value recursion construct that can be compiled via in-place update. The answer is no, because of the requirement that the sizes (of definitions that are forward-referenced) be known exactly at compile-time. In a context of separate compilation and higher-order functions, often the only thing that the compiler knows about definitions is their static types. With some data representation strategies, the sizes are functions of the static types, but not with other strategies. For example, the closures that represent function values can either follow a "two-block" strategy (a closure is a pair of a code pointer and a pointer to a separately-allocated block holding the values of free variables) or a "one-block" strategy (the code pointer and the values of the free variables are in the same block). With the two-block strategy, all definitions of function type $\tau_1 \to \tau_2$ have known size 2; but with the one-block strategy, the size is $1+n$ where n (the number of free variables) is not reflected in the function type and is therefore difficult to guess at compile-time.

There are several ways to relax the size requirement and therefore increase the usability of $\lambda_{\circ}$ as an intermediate language. First, one could permit values of size smaller than ex-

Variable:	$x \in \mathsf{vars}$		
Name:	$X \in \mathsf{names}$		
Expression:	$e \in \mathsf{expr} ::= x \mid \lambda x.e \mid e_1\ e_2$		λ-calculus
	$\mid \{r\} \mid e.X$		Record operations
	$\mid \mathtt{rec}\ b\ \mathtt{in}\ e$		Recursive definitions
Record row:	$r ::= \varepsilon \mid X = x, r$		
Binding:	$b ::= \varepsilon \mid x \diamond e, b$		
Size indication:	$\diamond ::= =_{[e]} \mid =_{[?]}$		

Fig. 29 Syntax of generalized $\lambda_\circ$

pected to fill the pre-allocated blocks. In this case, updating a pre-allocated block changes not only its contents but also its size, an operation that most memory managers support well. All we now need to determine statically is a conservative upper bound on the actual size. For example, if the type of a definition is a datatype (sum type), we can take the maximum of the sizes of its constructors. In the case of one-block closures, we can allocate dummy blocks with a fixed size, say 10 words, and instruct the compiler to never generate closures larger than this, switching to a two-block representation for closures with more than 9 free variables (such closures are uncommon). This simple extension can be formalized with minimal changes to $\lambda_\circ$, λ_a and the proofs presented in this paper.

Another way to relax the size requirement is to notice that the sizes of pre-allocated blocks do not need to be compile-time constants: the in-place update scheme works just as well if these sizes are determined by run-time computations that take place before the recursive definition is evaluated. For example, in the encoding of mixins outlined in Section 2.4.2, each component of a mixin could be represented not just as a generator function f, but as a pair (n, f) where n is the size of the result of f. The recursive definition implementing the `close` operation could, then, extract these sizes n from the run-time representation of the mixin and use them to pre-allocate dummy blocks.

In preparation for future work, we now sketch an extension of $\lambda_\circ$ where the size indications over bindings are no longer compile-time constants but arbitrary expressions. Figure 29 gives the syntax of this extended language. In bindings, the size indications are all evaluated before the evaluation of definitions begins, and cannot refer to the recursively defined variables.

From the standpoint of compilation, we believe that in-place update applies straightforwardly. However, a serious issue with this extension is how to ensure statically that the predicted sizes are correct: given a definition $x =_{[e_1]} e_2$, we would like to guarantee that e_2 will evaluate to a value of size the value of e_1. If e_1 is an arbitrary expression, a type system or another static analysis can not try and evaluate e_1 because this would make it undecidable. Instead, we have to find static means of ensuring the validity of definitions in the useful cases.

For this, we plan to start from Hirschowitz's type system for $\lambda_\circ$ [12] and extend it with dependent product types and a special sized type $\mathtt{Sized}_v(\tau)$, denoting the set of values of type τ and of size v. Given n and e of size n, one could give a dependent product type to the pair (n, e), namely $\langle x : int, \mathtt{Sized}_x(\tau)\rangle$. Conversely, take for example a dependent pair e of type $\langle x : int, \tau_1 \to \mathtt{Sized}_x(\tau_2)\rangle$, the expression $(\mathtt{snd}(e)\ e')$ has size $\mathtt{fst}(e)$, and this can be checked statically. This guarantees that the definition $x =_{[\mathtt{fst}(e)]} (\mathtt{snd}(e)\ e')$ is correct w.r.t. sizes.

Acknowledgements The authors warmly thank the anonymous referees for their detailed comments and helpful suggestions for improving the presentation.

References

1. Appel, A.W.: Compiling with continuations. Cambridge University Press (1992)
2. Ariola, Z.M., Blom, S.: Skew confluence and the lambda calculus with letrec. Annals of pure and applied logic **117**(1–3), 95–178 (2002)
3. Boudol, G.: The recursive record semantics of objects revisited. Journal of Functional Programming **14**(3), 263–315 (2004)
4. Boudol, G., Zimmer, P.: Recursion in the call-by-value lambda-calculus. In: Z. Ésik, A. Ingólfsdóttir (eds.) Fixed Points in Computer Science, *BRICS Notes Series*, vol. NS-02-2, pp. 61–66 (2002)
5. Cardelli, L.: A semantics of multiple inheritance. In: G. Kahn, D. MacQueen, G. Plotkin (eds.) Semantics of Data Types, *Lecture Notes in Computer Science*, vol. 173, pp. 51–67. Springer (1984)
6. Cousineau, G., Curien, P.L., Mauny, M.: The categorical abstract machine. Science of Computer Programming **8**(2), 173–202 (1987)
7. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: Programming Language Design and Implementation, pp. 50–63. ACM Press (1999)
8. Dreyer, D.: A type system for well-founded recursion. In: Principles of Programming Languages, pp. 293–305. ACM Press (2004)
9. Dreyer, D.: A type system for recursive modules. In: International Conference on Functional Programming, pp. 289–302. ACM Press (2007)
10. Erkök, L., Launchbury, J.: Recursive monadic bindings. In: International Conference on Functional Programming, pp. 174–185. ACM Press (2000)
11. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Third Edition. Prentice-Hall (2005)
12. Hirschowitz, T.: Mixin modules, modules, and extended recursion in a call-by-value setting. Ph.D. thesis, University of Paris VII (2003)
13. Hirschowitz, T.: Rigid mixin modules. In: Y. Kameyama, P. Stuckey (eds.) Functional and Logic Programming, *Lecture Notes in Computer Science*, vol. 2998, pp. 214–228. Springer (2004)
14. Hirschowitz, T., Leroy, X.: Mixin modules in a call-by-value setting. ACM Transactions on Programming Languages and Systems **27**(5), 857–881 (2005)
15. Hirschowitz, T., Leroy, X., Wells, J.B.: Compilation of extended recursion in call-by-value functional languages. In: Principles and Practice of Declarative Programming, pp. 160–171. ACM Press (2003)
16. Hirschowitz, T., Leroy, X., Wells, J.B.: Call-by-value mixin modules: Reduction semantics, side effects, types. In: D.A. Schmidt (ed.) Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, *Lecture Notes in Computer Science*, vol. 2986, pp. 64–78. Springer (2004)
17. Kelsey, R., Clinger, W.D., Rees, J.: The revised^5 report on the algorithmic language Scheme. SIGPLAN Notices **33**(9), 26–76 (1998)
18. Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., Adams, N.: ORBIT: An optimizing compiler for Scheme. In: Symposium on Compiler Construction, pp. 219–233. ACM Press (1986)
19. Lang, F., Dougherty, D., Lescanne, P., Rose, K.: Addressed term rewriting systems. Research report RR1999-30, ENS Lyon – LIP (1999)
20. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.10, documentation and user's manual. Available on the Web, `http://caml.inria.fr/pub/docs/manual-ocaml/` (2007)
21. Leroy, X., Doligez, D., Garrigue, J., Vouillon, J.: The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/` (1996–2008)
22. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (revised). The MIT Press (1997)
23. Moggi, E., Sabry, A.: An abstract monadic semantics for value recursion. Theoretical Informatics and Applications **38**(4), 375–400 (2004)
24. Nordlander, J., Carlsson, M., Gill, A.: Unrestricted call-by-value recursion. In: ACM SIGPLAN Workshop on ML. ACM Press (2008). To appear
25. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries, the revised report. Cambridge University Press (2003)
26. Plotkin, G.D.: Call-by-name, call-by-value and the λ-calculus. Theoretical Computer Science **1**(2), 125–159 (1975)

27. Russo, C.V.: Recursive structures for Standard ML. In: International Conference on Functional Programming, pp. 50–61. ACM Press (2001)
28. Sewell, P., Stoyle, G., Hicks, M., Bierman, G., Wansbrough, K.: Dynamic rebinding for marshalling and update, via redex-time and destruct-time reduction. Journal of Functional Programming **18**(4), 437–502 (2008)
29. Shivers, O.: Control-flow analysis in Scheme. In: Programming Language Design and Implementation 1988, pp. 164–174. ACM Press (1988)
30. Syme, D.: Initializing mutually referential abstract objects: The value recursion challenge. In: Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005), *Electronic Notes in Theoretical Computer Science*, vol. 148(2), pp. 3–25. Elsevier (2006)
31. Waddell, O., Sarkar, D., Dybvig, R.K.: Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. Higher-Order and Symbolic Computation **18**(3/4), 299–326 (2005)
32. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1), 38–94 (1992)