Toward Flexible Equational Reasoning for a Purely Imperative Calculus (final report of *stage*)

Project supervisor: J. B. WELLS
Student: Benoit DELAHAYE

2005-09-12T04:05

Abstract

This is the final report of the *stage* of Benoit Delahaye, carried out under the supervision of J. B. Wells while visiting Heriot-Watt University.

The *stage* focused on the i-calculus, a new proposal by J. B. Wells for reasoning about the equivalence of computer programs in an imperative language. The work was to flesh out the details and definitions of the i-calculus, debug its design, and make excellent progress toward filling in the proofs of its properties. The i-calculus ("i" for "imperative") is a *purely imperative* language with explicit memory allocation, garbage collection, pointers, address arithmetic, assignments, conditional and unconditional jumps, and pointers to blocks of instructions. The i-calculus is *missing* any notion of functions, parameter passing, or returning of values; such higher-level features must be encoded into its more basic features. Despite its primitive nature, the i-calculus is equipped with a rich equational theory of program equivalences. Notably, we present a translation Tv (a fairly normal continuation-passing-style compilation) from a high-level call-by-value λ-calculus with tuples, tagged variants, and integers such that if $E \to E'$ by the rewriting rules of this λ-calculus, then $\mathsf{Tv}(E) \twoheadrightarrow \mathsf{Tv}(E')$ by i-calculus rewriting. We define an operational semantics for the i-calculus by choosing a rewriting strategy, and present our work toward proving the rewriting rules sound, i.e., that non-evaluation use of the rewriting rules never changes the observable evaluation behavior.

This work is not finished; the status is discussed within the paper.

1 Motivation, goal, and status

It is accepted wisdom that when compiling a functional language, there is an initial phase when the program can be represented in a λ-calculus-based language, and during this phase flexible equational reasoning can be used for optimizations, and there is a later phase when the program must be represented in a low-level, machine-oriented language, and in this phase flexible equational reasoning is not possible. In this paper we aim to undermine this accepted wisdom. We exhibit the i-calculus, which is a low-level, purely imperative, machine-oriented language, but nonetheless supports all the equational reasoning flexibility of the λ-calculus. We present our work toward proving the equational reasoning rules for the i-calculus are correct using the framework and methods in earlier work by Wells, Plump, and Kamareddine [1].

The proof of rewriting soundness is still in progress. We are confident that it is proceeding smoothly and foresee no serious obstacles. Although Delahaye's *stage* is now finished, we plan to finish the work with additional collaboration including further visits. We have strong hopes

that we can submit a short version to ESOP '06 (the European Symposium on Programming) on 2005-10-14, and that this will have a good chance of being accepted.

2 The i-calculus

This section defines the i-calculus ("i" for "imperative"), a *purely imperative* language. It is a primitive, machine-oriented language at roughly the level of assembly language, although we will show in this paper that it has the same equational reasoning power as a high-level λ-calculus.

2.1 Bases and locations

Figure 1 defines *bases* and *locations*. Bases roughly correspond to pointers to the beginning of memory allocation chunks, while locations correspond roughly to offsets within the chunks. This correspondence is not precise, because bases and locations (especially those of size 1) can sometimes be considered to correspond to machine registers.

The binding form for bases, the alloc operator, associates with each base a maximum offset, a number in $\{0, 1, 2, \ldots\}$. This corresponds to the size of a memory allocation.

Members of StaticLocation stand for locations which are allowed to be modified at most once, while members of DynamicLocation may be modified as many times as desired. If an attempt is made to write to a static location more than once, or to read from it before it is initialized, the program becomes undefined. In an actual implementation, of course such operations would most likely simply proceed (with possibly garbage results), but the official semantics of such a program is that it is undefined and allowed to do anything. This helps to justify that certain program transformations are meaning preserving.

2.2 Grammar and metavariable conventions

Figure 1 also gives the grammar and metavariable conventions of the i-calculus. The i-calculus has explicit memory allocation (alloc X) and garbage collection, pointers (ℓ) and pointer dereferencing ($!\ell$), integers and basic arithmetic operations, address arithmetic ($x.o\ A{=}\ i$), assignments ($\ell{:=}V$), conditional (bnz $i\ P$) and unconditional (jmp P) jumps, and *code closures* ($[P]$). An i-calculus program is a sequence of operations, i.e., an operation followed by a program (O; P).

Groups of assignments can be collected together in *transformers* (T) for ease of equational reasoning. When an i-calculus program of the form alloc X; T; O; P is being executed, one can think of the indicated occurrences of X and T as representing the machine memory state (while any occurrences of X and T inside P still represent instructions to be executed), the O as representing the instruction currently pointed to by the program counter register, and P as representing the subsequent instructions. Although execution of i-calculus programs is defined by rewriting rules that transform programs to programs, in fact it can be seen as merely a notational variation of a fairly normal abstract machine definition of execution.

Code closures are the main place where the i-calculus differs from an assembly language. If code closures do not refer to allocation bases that are bound by alloc in an enclosing location in the program, then an i-calculus program can be seen as an instance of assembly language where all free bases correspond to machine registers. When code closures do have free bases that are bound in the enclosing context, then a code closure $[P]$ must be seen as standing for a sequence of machine instructions that allocates memory and stores in it the free bases of P together with a code pointer that points to the instructions corresponding to P prefixed by instructions for extracting the free bases from the closure and arranging them in the right

registers. The way that i-calculus code closures abstract away from this memory management detail is one of the keys to the flexible equational reasoning of the i-calculus.

Figure 1 also defines *dereferencing contexts* and *substitution contexts*. Dereferencing contexts are used to formulate the many rewriting rules that execute i-calculus programs. In contrast, substitution contexts are used solely in the substitution rewriting rule, which does not correspond to any execution step but instead gives the i-calculus the same equational reasoning flexibility as the λ-calculus.

y, z	$\in$ StaticBase		
a, b, k	$\in$ DynamicBase		
x	$\in$ Base	$=$	StaticBase $\cup$ DynamicBase
o	$\in$ Offset	$=$	NaturalNumbers $= \{0, 1, 2, \ldots\}$
$y.o$	$\in$ StaticLocation	$=$	StaticBase $\times$ Offset
$b.o$	$\in$ DynamicLocation	$=$	DynamicBase $\times$ Offset
ℓ	$\in$ Location	$=$	StaticLocation $\cup$ DynamicLocation
i	$\in$ Integer		
P	$\in$ Program	$::=$	$O; P \mid$ undef
O	$\in$ Operation	$::=$	$S \mid J$
B	$\in$ BasicStoreOp	$::=$	alloc $X \mid V := W \mid T$
S	$\in$ StoreOp	$::=$	$V\ A{=}\ W \mid V := !(W) \mid B$
A	$\in$ ArithFun	$::=$	$+ \mid - \mid \cdots$
J	$\in$ Jump	$::=$	jmp $W \mid$ bnz $W_1\ W_2$
X	$\in$ Alloc	$::=$	$\{x_1(o_1), \ldots, x_n(o_n)\}$ where $x_j \neq x_k$ if $j \neq k$
T	$\in$ StoreTransformer	$::=$	$\{\ell_1 := W_1, \ldots, \ell_n := W_n\}$ where $\ell_i \neq \ell_j$ if $i \neq j$
V	$\in$ BasicValue	$::=$	$\ell \mid !\ell \mid [P] \mid i \mid$ junk
U	$\in$ ComplexValue	$::=$	$V \mid B; U$
W	$\in$ SafeValue $\subset$ ComplexValue		(defined in section 2.3)
$\overline{D}$	$\in$ DereferencingCtxt	$::=$	$\Box := V \mid \ell := \Box$
			$\mid \Box := !(V) \mid \ell := !(\Box)$
			$\mid \Box\ A{=}\ V \mid \ell\ A{=}\ \Box$
			$\mid$ jmp $\Box \mid$ bnz $\Box\ V \mid$ bnz $i\ \Box$
$\widehat{P}$	$\in$ SubstitutionCtxt	$::=$	$\widehat{O}; P \mid O; \widehat{P}$
$\widehat{O}$		$::=$	$\widehat{S} \mid \widehat{J}$
$\widehat{B}$		$::=$	$V := \widehat{U} \mid \widehat{T}$
$\widehat{S}$		$::=$	$V\ A{=}\ \widehat{U} \mid V := !(\widehat{U}) \mid \widehat{B}$
$\widehat{J}$		$::=$	jmp $\widehat{U} \mid$ bnz $\widehat{U}\ W \mid$ bnz $W\ \widehat{U}$
$\widehat{T}$		$::=$	$\{\ell_1 := W_1, \ldots, \ell_k := \widehat{U_k}, \ldots, \ell_n := W_n\}$
$\widehat{V}$		$::=$	$[\widehat{P}]$
$\widehat{U}$		$::=$	$\widehat{V} \mid B; \widehat{U} \mid \widehat{B}; W \mid \Box$

Figure 1: The syntax and metavariable conventions of the i-calculus

2.3 Effects and safe values

Figure 1 refers to a set SafeValue. The definition of this set is too complex to be easily defined merely with a grammar, so instead we define the syntactic category ComplexValue and filter out the undesirable members. In the following rules, the judgement $U \diamond (\mathcal{R}, \mathcal{W})$ should be read as meaning the following:

> "When the instructions in U are executed, they only allocate, read, and write static locations. $\mathcal{R}$ is the set of locations read by U which are not initialized by U. $\mathcal{W}$ is the set of locations written by U that are not allocated by U. The instructions of U do not write to any location more than once, nor do they write to an invalid offset of a base they allocate, nor do they read a location which they allocate without initializing, nor do they write a location after reading it. For any U' nested inside a T inside U, it holds that U' allocates all the locations it writes."

The rules for $U \diamond (\mathcal{R}, \mathcal{W})$ are as follows:

$V \diamond (\varnothing, \varnothing)$ if $V \neq !\ell$

$!y.o \diamond (\{y.o\}, \varnothing)$

$$\frac{U \diamond (\mathcal{R}, \mathcal{W});\ (U_i \diamond (\mathcal{R}_i, \varnothing) \text{ for } i \in \{1,\ldots,n\});\ \{\ell_1,\ldots,\ell_n\} \cap \mathcal{W} = \varnothing;\ (\mathcal{R}_i \cap \mathcal{W} = \varnothing \text{ for } i \in \{1,\ldots,n\});\ (\mathcal{R}_i \cap \{\ell_1,\ldots,\ell_n\} = \varnothing \text{ for } i \in \{1,\ldots,n\});\ \{\ell_1,\ldots,\ell_n\} \subset \mathsf{StaticLocation}}{\{\ell_1 := U_1,\ \ldots,\ \ell_n := U_n\};\ U \diamond ((\mathcal{R} \setminus \mathcal{W}) \cup \mathcal{R}_1 \cup \ldots \cup \mathcal{R}_n, \{\ell_1,\ldots,\ell_n\} \cup \mathcal{W})}$$

$$\frac{U \diamond (\mathcal{R}, \mathcal{W});\ (\forall y.o \in \mathcal{R}.\ y \notin \{y_1,\ldots,y_n\});\ (\forall y.o \in \mathcal{W}.\ y = y_i \Longrightarrow o \in \{0,\ldots,o_i\})}{\mathsf{alloc}\ \{y_1(o_1),\ \ldots,\ y_n(o_n)\};\ U \diamond (\mathcal{R}, \{\, y.o \in \mathcal{W} \,|\, y \notin \{y_1,\ldots,y_n\}\,\})}$$

$$\frac{\{\ell := U_1\};\ U_2 \diamond (\mathcal{R}, \mathcal{W})}{\ell := U_1;\ U_2 \diamond (\mathcal{R}, \mathcal{W})}$$

Finally, we define that $U \in \mathsf{SafeValue}$ iff $U \diamond (\mathcal{R}, \varnothing)$ for some $\mathcal{R}$.

A safe value $W \in \mathsf{SafeValue}$ is a basic value together with a sequence of instructions to allocate and initialize memory that it can depend on. Because a safe value behaves nicely and its side effects are encapsulated, it can be treated like a semantic value. The substitution rewriting rule takes advantage of this to substitute safe values freely without changing the meaning of the program.

2.4 Free bases, α-conversion, and syntactic conventions

We define the *free bases* of P, written $\mathsf{FB}(P)$, as follows:

$$\begin{aligned}
&\mathsf{FB}(\mathsf{undef}) = \mathsf{FB}(i) = \mathsf{FB}(\mathsf{junk}) = \varnothing \\
&\mathsf{FB}(x.o) = \mathsf{FB}(!x.o) = \{x\} \\
&\mathsf{FB}(\mathsf{jmp}\ W) = \mathsf{FB}(W) \\
&\mathsf{FB}(\mathsf{bnz}\ W_1\ W_2) = \mathsf{FB}(W_1) \cup \mathsf{FB}(W_2) \\
&\mathsf{FB}(V := W) = \mathsf{FB}(V\ A{=}\ W) = \mathsf{FB}(V := !(W)) = \mathsf{FB}(V) \cup \mathsf{FB}(W) \\
&\mathsf{FB}(\{\ell := W\} \cup T) = \mathsf{FB}(\ell) \cup \mathsf{FB}(W) \cup \mathsf{FB}(T) \\
&\mathsf{FB}([P]) = \mathsf{FB}(P) \\
&\mathsf{FB}(\mathsf{alloc}\ X;\ P) = (\mathsf{FB}(P) \setminus \mathsf{Dom}(X)) \\
&\mathsf{FB}(O;\ P) = \mathsf{FB}(O) \cup \mathsf{FB}(P) \text{ if } O \neq \mathsf{alloc}\ X
\end{aligned}$$

We define also the convenience function $\overline{\mathsf{FB}}$ on sets of locations so that $\overline{\mathsf{FB}}(\{x_1.o_1,\ldots,x_n.o_n\}) = \{x_1,\ldots,x_n\}$.

We will introduce the usual notion of α-conversion as follows. Let $(x\,x')\,P$ mean the result of replacing in P every occurrence of x by x' and vice versa. We retroactively modify our definitions by declaring that, if $\{x,x'\} \cap \mathsf{FB}(P) = \varnothing$, then $(x\,x')\,P = P$. Thus, after this point in the paper, all references to a P or some piece of syntax which can occur inside P can be thought of as referring to an equivalence class of syntactic entities from the definitions before this point.

Every program ends with undef to represent that a program which runs off the end of its instruction sequence will do something undefined. However, this is tedious to write, so in examples we allow writing "jmp V; undef" as "jmp V", because in that case undef will not be reached.

Because bases with size 1 will occur frequently, we allow writing "$x(0)$" and "$x.0$" as "x".

Note that although we use the notation of mathematical sets when writing operations on transformers ($T \in \mathsf{StoreTransformer}$), in fact transformers are not sets but lists, i.e., the order of their components is significant. So wherever there is set notation used to specify a transformer, the reader should substitute in their mind similar (but more verbose) operations on lists that yield deterministic results. The only reason we need the order to matter is because it allows making the evaluation step relation deterministic, and this reduces the number of cases that must be considered in the proof that the equational theory of the i-calculus is correct. We believe the proof could be enhanced to work with transformers as sets, but it would be tedious to do so.

2.5 Examples of programs and operations in i-calculus

We will show here how the i-calculus works with some basic examples. As explained earlier, a program which does not end up with a jump to some context has an undefined behavior. In these examples, k represents the continuation of the current execution.

- Memory allocation and initialization of a static value :

$$\mathsf{alloc}\ \{y(0)\};\ \{y.0 := 2\};\ \mathsf{jmp}\ !k$$

- We can also allocate tables of static/dynamic values and modify one dynamic value with the contents of a static one :

$$\begin{array}{l}\mathsf{alloc}\ \{y(3),a(2)\};\\ \{y.0 := 0, y.1 := 1, y.2 := 2, y.3 := 3, a.0 := 4, a.1 := 5, y.2 := 6\};\\ a.1 := !y.2;\ \mathsf{jmp}\ !k\end{array}$$

- It is possible to do some address arithmetic. For example if the content of $a.0$ above is a pointer to $y.0$, we can change it directly to a pointer to $y.2$ (N.B. any value can remain uninitialized) :

$$\begin{array}{l}\mathsf{alloc}\ \{y(3),a(2)\};\\ \{y.0 := 0, y.1 := 1, y.2 := 2, y.3 := 3, a.0 := y.0\};\ a.0 \mathrel{+}= 2;\ \mathsf{jmp}\ !k\end{array}$$

- There can be multiple dereferencing of pointers. For example if $a.0$ is a pointer to $a.2$, which itself is a pointer to $y.0$, we can put the value contained in $y.0$ in $a.1$ by doing this :

$$\begin{array}{l}\mathsf{alloc}\ \{y(3),a(2)\};\ \{y.0 := 0, y.1 := 1, y.2 := 2, y.3 := 3, a.0 := a.2, a.2 := y.0\};\\ a.1 := !(!a.0);\ \mathsf{jmp}\ !k\end{array}$$

- One more possible thing would be conditional and unconditional jumps. For example to jump to a piece of code contained in $y.3$ if $a.0$ is equal to $a.1$ or to another piece of code, contained in $y.2$ if they are different. We will need for this another dynamic (or static) variable (N.B. $a -= i$ can also do classical arithmetic operations if a contains a value and not a pointer) :

$$\begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.0 := 0, y.1 := 1, y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2\};\\ b.0 := !a.0;\ b.0 -= !a.1;\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

- Finally, an example using the safe values defined earlier. We allocate and initialize some memory inside a value :

$$\mathsf{alloc}\ \{a(0)\};\ \{a.0 := (\mathsf{alloc}\ \{y(0)\};\ \{y.0 := 3\};\ !y.0)\};\ a.0 := 5;\ \mathsf{jmp}\ !k$$

2.6 Definitions used in the rewriting rules

We now define auxiliary functions that will be used in the definition of the rewriting rules.

- $\mathsf{Dom}(X) = \{x \mid x(o) \in X\}$
- $\mathsf{Dom}(T) = \{\ell_i \mid (\ell_i := W_i) \in T\}$
- $\mathsf{Ran}(T) = \{W_i \mid (\ell_i := W_i) \in T\}$
- $T \setminus \ell = \{(\ell_i := W_i) \in T \,|\, \ell_i \neq \ell\}$
- $T(\ell_i) = \begin{cases} W_i & \text{if } (\ell_i := W_i) \in T \\ \text{undefined} & \text{otherwise} \end{cases}$
- $\mathsf{ComposeUndef}(T, T') \iff \mathsf{Dom}(T) \cap \mathsf{Dom}(T') \cap \mathsf{StaticLocation} \neq \varnothing$
- $\mathsf{ComposeBlocked}(T, T') \iff \exists \ell \in \mathsf{Dom}(T) \cap \mathsf{Dom}(T') \cap \mathsf{DynamicLocation}.T(\ell) \notin \mathsf{BasicValue}$
- $$\begin{array}{lcl} !\ell\langle T\rangle & = & \begin{cases} !\ell & \text{if } \ell \notin \mathsf{Dom}(T) \\ T(\ell) & \text{otherwise} \end{cases} \\ V\langle T\rangle & = & V \text{ if } V \text{ not of form } !\ell \\ (\mathsf{alloc}\ X; U)\langle T\rangle & = & \mathsf{alloc}\ X;\ (U\langle T\rangle) \text{ where } \mathsf{Dom}(X) \cap \mathsf{FB}(T) = \varnothing \\ (T_1; U)\langle T_2\rangle & = & \{\ell := (T_1(\ell))\langle T_2\rangle \,|\, \ell \in \mathsf{Dom}(T_1)\};\ (U\langle T_2\rangle) \quad \text{where} \quad \overline{\mathsf{FB}}(\mathsf{Dom}(T_1)) \cap \\ & & \mathsf{FB}(T_2) = \varnothing \\ (y.o := W; U)\langle T\rangle & = & y.o := (W\langle T\rangle);\ (U\langle T\rangle) \text{ where } y \notin \mathsf{FB}(T) \end{array}$$
- $\mathsf{Compose}(T, T') = \{\ell := (T'(\ell))\langle T\rangle \,|\, \ell \in \mathsf{Dom}(T')\} \cup \{(\ell := W) \in T \,|\, \ell \notin \mathsf{Dom}(T')\}$
- $$\begin{array}{lcl} \mathsf{Bind}(\mathsf{alloc}\ X) & = & \mathsf{Dom}(X) \\ \mathsf{Bind}(O) & = & \varnothing \text{ if } O \neq \mathsf{alloc}\ X \end{array}$$

- $\mathsf{Capture}(O; \widehat{P}) = \mathsf{Bind}(O) \cup \mathsf{Capture}(\widehat{P})$
  $\mathsf{Capture}(\square) = \varnothing$
  $\mathsf{Capture}([\widehat{P}]) = \mathsf{Capture}(\widehat{P})$
  $\mathsf{Capture}(\mathsf{jmp}\ \widehat{U}) = \mathsf{Capture}(\mathsf{bnz}\ W\ \widehat{U}) = \mathsf{Capture}(\mathsf{bnz}\ \widehat{U}\ W) = \mathsf{Capture}(V := \widehat{U})$
  $= \mathsf{Capture}(V\ A = \widehat{U}) = \mathsf{Capture}(V := !(\widehat{U})) = \mathsf{Capture}(\widehat{U})$
  $\mathsf{Capture}(\{\ell_1 := W_1, \ldots, \ell_k := \widehat{U_k}, \ldots, \ell_n := W_n\}) = \mathsf{Capture}(\widehat{U_k})$
  $\mathsf{Capture}(\widehat{O}; P) = \mathsf{Capture}(\widehat{O}; U) = \mathsf{Capture}(\widehat{O})$
  $\mathsf{Capture}(O; \widehat{U}) = \mathsf{Bind}(O) \cup \mathsf{Capture}(\widehat{U})$

2.7 Rewriting rules

We present here the rewriting rules of the i-calculus. All the rewriting rules defined here transform programs ($P \in \mathsf{Program}$) to programs. The rules may be used wherever a P occurs inside another P'.

A particular strategy of using the rules will be defined later to give an operational semantics for the i-calculus. Our overall goal will be to show that the rules given here provide a powerful and flexible equational theory, while at the same time showing that they are correct with respect to the operational semantics, i.e., any use of these rules at any location in a program preserves observable behavior of the program.

As our proof is not yet finished, the definitions of the rewriting rules might still change in some small ways to simplify the proof or to take care of some problems we have not thought of yet.

We will present *rule schemes* in the form "$\dot{R} : \dot{P}_1 \to \dot{P}_2$ if $\dot{C}$", where $\dot{R}$ is a *rule name scheme*, $\dot{P}_1$ and $\dot{P}_2$ are *program schemes*, and $\dot{C}$ is a *condition scheme*. $\dot{R}$, $\dot{P}_1$, $\dot{P}_2$, and $\dot{C}$ may mention metavariables defined in this paper. When θ is a valuation that maps metavariables to members of the appropriate sets, and $C = \theta(\dot{C})$ is a true statement, then $R = \theta(\dot{R})$ is a *rule* that maps the program $\theta(\dot{P}_1)$ to the program $\theta(\dot{P}_2)$. If $\dot{C}$ is simply "true", then the "if $\dot{C}$" portion may be omitted.

Given a rule scheme "$\dot{R} : \dot{P}_1 \to \dot{P}_2$ if $\dot{C}$", we allow using the rule $R = \theta(\dot{R})$ as a function which is defined by all of program pairs formed by considering all the valuations θ' that make $\theta'(\dot{C})$ true and $\theta'(\dot{R}) = R$. That is, if $R = \theta(\dot{R})$, then the rule R is the function on programs such that $R(P_1) = P_2$ iff $R = \theta'(\dot{R})$ for some θ' where $P_1 = \theta'(\dot{P}_1)$, $P_2 = \theta'(\dot{P}_2)$, and $\theta'(\dot{C})$ is true. We have been careful to define the rule schemes such that each rule always yields a well defined function.

For ease of understanding, we divide the rules into categories:

- Sequencing store effects :

S_1 :	$\mathsf{alloc}\ X_1; \mathsf{alloc}\ X_2; P$	$\longrightarrow$	$\mathsf{alloc}\ X_1 \cup X_2; P$ if $(\mathsf{Dom}(X_1) \cap \mathsf{Dom}(X_2) = \varnothing)$
S_2 :	$T_1; T_2; P$	$\longrightarrow$	$\mathsf{Compose}(T_1, T_2); P$ if $\neg(\mathsf{ComposeUndef}(T_1, T_2)$ or $\mathsf{ComposeBlocked}(T_1, T_2))$
S_3 :	$T; \mathsf{alloc}\ X; P$	$\longrightarrow$	$\mathsf{alloc}\ X; T; P$ if $(\mathsf{Dom}(X) \cap \mathsf{FB}(T) = \varnothing)$
S_4 :	$\ell := W; P$	$\longrightarrow$	$\{\ell := W\}; P$
S_5 :	$\ell_1 := !(\ell_2); P$	$\longrightarrow$	$\{\ell_1 := !\ell_2\}; P$
S_6 :	$\overline{D}[S_1; \ldots; S_n; V]; P$	$\longrightarrow$	$S_1; \ldots; S_n; \overline{D}[V]; P$
$\mathsf{S}_{7,\ell}$:	$\{\ell := (S_1; \ldots; S_n; V)\} \uplus T; P$	$\longrightarrow$	$S_1; \ldots; S_n; \{\ell := V\} \uplus T; P$

- Dereferencing :

$$
\begin{array}{lllll}
\mathrm{D}_4: & T; \overline{D}[!x.o]; P & \longrightarrow & T; \overline{D}[V]; P & \\
 & & & \text{if } T(x.o) = V & \\
\mathrm{D}_{1,\ell}: & \mathsf{alloc}\ \{a(o)\} \cup X; \{\ell := !a.o'\} \uplus T; P & \longrightarrow & \mathsf{alloc}\ \{a(o)\} \cup X; \{\ell := \mathsf{junk}\} \uplus T; P & \\
 & & & \text{if } 0 \leq o' \leq o & \\
\mathrm{D}_2: & \mathsf{alloc}\ \{a(o)\} \cup X; \overline{D}[!a.o']; P & \longrightarrow & \mathsf{alloc}\ \{a(o)\} \cup X; \overline{D}[\mathsf{junk}]; P & \\
 & & & \text{if } 0 \leq o' \leq o & \\
\mathrm{D}_3: & \mathsf{alloc}\ \{a(o)\} \cup X; T; \overline{D}[!a.o']; P & \longrightarrow & \mathsf{alloc}\ \{a(o)\} \cup X; T; \overline{D}[\mathsf{junk}]; P & \\
 & & & \text{if } 0 \leq o' \leq o \text{ and } a.o' \notin \mathsf{Dom}(T) &
\end{array}
$$

- Control :

$$
\begin{array}{llllll}
\mathrm{C}_1: & \mathsf{jmp}\ [P]; P' & \longrightarrow & P & & \\
\mathrm{C}_2: & \mathsf{bnz}\ i\ [P]; P' & \longrightarrow & P & \text{if } i \neq 0 & \\
\mathrm{C}_3: & \mathsf{bnz}\ i\ [P]; P' & \longrightarrow & P' & \text{if } i = 0 &
\end{array}
$$

- Arithmetic :

$$
\begin{array}{llll}
\mathrm{A}_1: & T \uplus \{a.o := i\}; a.o\ A = i'; P & \longrightarrow & T \uplus \{a.o := i\,A\,i'\}; P \\
\mathrm{A}_2: & T \uplus \{a.o := x.i\}; a.o\ A = i'; P & \longrightarrow & T \uplus \{a.o := x.(i\,A\,i')\}; P
\end{array}
$$

- Garbage collection :

$$
\begin{array}{llllll}
\mathrm{G}_{1,X_2}: & \mathsf{alloc}\ X_1 \uplus X_2; P & \longrightarrow & \mathsf{alloc}\ X_1; P & \text{if } \mathsf{Dom}(X_2) \cap \mathsf{FB}(P) = \varnothing & \\
\mathrm{G}_{3,T_2}: & \mathsf{alloc}\ X; T_1 \uplus T_2; P & \longrightarrow & \mathsf{alloc}\ X; T_1; P & \text{if } \overline{\mathsf{FB}}(\mathsf{Dom}(T_2)) \subseteq \mathsf{Dom}(X) \setminus & \\
 & & & & (\overline{\mathsf{FB}}(\mathsf{Ran}(T_1)) \cup \mathsf{FB}(P)) \text{ and } T_2 \neq & \\
 & & & & \varnothing \text{ and } \mathsf{Ran}(T_2) \subseteq \mathsf{BasicValue} &
\end{array}
$$

- Substitution :

$$
\mathrm{T}_{\widehat{P}}: \quad T; \widehat{P}[!y.o] \quad \longrightarrow \quad T; \widehat{P}[W] \quad \text{if } (T(y.o) = W) \wedge (\mathsf{Capture}(\widehat{P}) \cap (\mathsf{FB}(W) \cup \{y\}) = \varnothing)
$$

- Errors :

$$
\begin{array}{lllll}
\mathrm{E}_1: & V := W; P & \longrightarrow \ \mathsf{undef} & \text{if } (V \text{ of form } [P] \text{ or } i \text{ or junk}) \\
\mathrm{E}_2: & V\ A{=}\ W; P & \longrightarrow \ \mathsf{undef} & \text{if } (V \text{ of form } [P] \text{ or } i \text{ or junk}) \\
 & & & \vee\, (W \text{ of form } \ell \text{ or } [P] \text{ or junk}) \\
\mathrm{E}_3: & V := !(W); P & \longrightarrow \ \mathsf{undef} & \text{if } (V \text{ of form } [P] \text{ or } i \text{ or junk}) \\
 & & & \vee\, (W \text{ of form } [P] \text{ or } i \text{ or junk}) \\
\mathrm{E}_4: & \mathsf{jmp}\, W; P & \longrightarrow \ \mathsf{undef} & \text{if } (W \text{ of form } \ell \text{ or } i \text{ or junk}) \\
\mathrm{E}_5: & \mathsf{bnz}\, W\ W'; P & \longrightarrow \ \mathsf{undef} & \text{if } (W \text{ of form } \ell \text{ or } [P] \text{ or junk}) \\
 & & & \vee\, (W' \text{ of form } \ell \text{ or } i \text{ or junk}) \\
\mathrm{E}_6: & \mathsf{alloc}\, \{x(o)\} \uplus X; \{x.o' := W\} \uplus T; P & \longrightarrow \ \mathsf{undef} & \text{if } o' \notin \{0,\ldots,o\} \\
\mathrm{E}_7: & \mathsf{alloc}\, \{x(o)\} \uplus X; \{\ell := !x.o'\} \uplus T; P & \longrightarrow \ \mathsf{undef} & \text{if } o' \notin \{0,\ldots,o\} \\
\mathrm{E}_8: & \mathsf{alloc}\, \{x(o)\} \uplus X; T; \overline{D}[!x.o']; P & \longrightarrow \ \mathsf{undef} & \text{if } o' \notin \{0,\ldots,o\} \\
\mathrm{E}_9: & S; \mathsf{undef} & \longrightarrow \ \mathsf{undef} & \\
\mathrm{E}_{10}: & y.o\ A{=}\ W; P & \longrightarrow \ \mathsf{undef} & \\
\mathrm{E}_{11}: & T; T'; P & \longrightarrow \ \mathsf{undef} & \text{if } \mathsf{ComposeUndef}(T, T') \\
\mathrm{E}_{12}: & \mathsf{alloc}\, \{y(o)\} \cup X; \{\ell := !y.o'\} \uplus T; P & \longrightarrow \ \mathsf{undef} & \\
\mathrm{E}_{13}: & \mathsf{alloc}\, \{y(o)\} \cup X; \overline{D}[!y.o']; P & \longrightarrow \ \mathsf{undef} & \\
\mathrm{E}_{14}: & \mathsf{alloc}\, \{y(o)\} \cup X; T; \overline{D}[!y.o']; P & \longrightarrow \ \mathsf{undef} & \text{if } y.o' \notin \mathsf{Dom}(T)
\end{array}
$$

2.8 Examples of rewriting i-calculus programs

Here are two examples showing how the rewriting rules work on some of the basic programs shown in section 2.5.

- This first example is a program meant to compare two values recorded in a dynamic memory, and jump to a program recorded in a static memory. Here is how the reduction goes on :

$$\begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.0 := 0, y.1 := 1, y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2\};\\ b.0 := !a.0;\ b.0 \mathrel{-}= !a.1;\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

We will start by dereferencing $!a.0$ after the transformer ;

$$\xrightarrow{\mathrm{D}_4}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.0 := 0, y.1 := 1, y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2\};\\ b.0 := 1;\ b.0 \mathrel{-}= !a.1;\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

then promote the store operation into a transformer ;

$$\xrightarrow{\mathrm{S}_5}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.0 := 0, y.1 := 1, y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2\};\\ \{b.0 := 1\};\ b.0 \mathrel{-}= !a.1;\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

merge the two transformers ;

$$\xrightarrow{\mathrm{S}_2}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.0 := 0, y.1 := 1, y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2, b.0 := 1\};\\ b.0 \mathrel{-}= !a.1;\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

dereference $!a.1$ after the transformer ;

$$\xrightarrow{\mathrm{D}_4}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.0 := 0, y.1 := 1, y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2, b.0 := 1\};\\ b.0 \mathrel{-}= 2;\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

execute the arithmetic operation ;

$$\xrightarrow{\mathrm{A}_1}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.0 := 0, y.1 := 1, y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2, b.0 := -1\};\\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

garbage collect the items not used in the transformer ($T_2 = \{y.0 := 0, y.1 := 1\}$) ;

$$\xrightarrow{\mathrm{G}_{3,T_2}}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2, b.0 := -1\};\\ \mathsf{bnz}\ !b.0\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

dereference $!b.0$;

$$\xrightarrow{\mathrm{D}_4}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2, b.0 := -1\};\\ \mathsf{bnz}\ -1\ !y.2;\ \mathsf{jmp}\ !y.3\end{array}$$

dereference $!y.2$;

$$\xrightarrow{\mathrm{D}_4}\quad \begin{array}{l}\mathsf{alloc}\ \{y(3), a(2), b(0)\};\\ \{y.2 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k],\\ \ \ y.3 := [\mathsf{alloc}\ \{y(0)\};\ y.0 := 4;\ \mathsf{jmp}\ !k], a.0 := 1, a.1 := 2, b.0 := -1\};\\ \mathsf{bnz}\ -1\ [\mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k];\ \mathsf{jmp}\ !y.3\end{array}$$

execute the conditional jump ;

$$\xrightarrow{\mathrm{C}_2}\quad \mathsf{alloc}\ \{y(0)\};\ y.0 := 1;\ \mathsf{jmp}\ !k$$

and finally promote the store operation into a transformer

$$\xrightarrow{\mathrm{S}_4}\quad \mathsf{alloc}\ \{y(0)\};\ \{y.0 := 1\};\ \mathsf{jmp}\ !k$$

which is a reasonable final state for our reduction, it shows the final state of the memory and does nothing but jump to a new context.

- This second example shows the use of safe values in a basic program that just allocates, initializes and modifies a dynamic memory.

  $$\mathsf{alloc}\ \{a(0)\};\ \{a.0 := (\mathsf{alloc}\ \{y(0)\};\ \{y.0 := 3\};\ !y.0)\};\ a.0 := 5;\ \mathsf{jmp}\ !k$$

  We start by getting rid of the safe value ;

  $$\xrightarrow{S_{7,a.0}}\quad \mathsf{alloc}\ \{a(0)\};\ \mathsf{alloc}\ \{y(0)\};\ \{y.0 := 3\};\ \{a.0 := !y.0\};\ a.0 := 5;\ \mathsf{jmp}\ !k$$

  then we merge the alloc 's ;

  $$\xrightarrow{S_1}\quad \mathsf{alloc}\ \{a(0), y(0)\};\ \{y.0 := 3\};\ \{a.0 := !y.0\};\ a.0 := 5;\ \mathsf{jmp}\ !k$$

  merge the transformers ;

  $$\xrightarrow{S_2}\quad \mathsf{alloc}\ \{a(0), y(0)\};\ \{y.0 := 3, a.0 := 3\};\ a.0 := 5;\ \mathsf{jmp}\ !k$$

  promote the store operation into a transformer ;

  $$\xrightarrow{S_4}\quad \mathsf{alloc}\ \{a(0), y(0)\};\ \{y.0 := 3, a.0 := 3\};\ \{a.0 := 5\};\ \mathsf{jmp}\ !k$$

  and again merge the transformers taking care of keeping the right value for $a.0$.

  $$\xrightarrow{S_2}\quad \mathsf{alloc}\ \{a(0), y(0)\};\ \{y.0 := 3, a.0 := 5\};\ \mathsf{jmp}\ !k$$

  This is once again a reasonable final state for evaluation.

3 Translating a high-level λ-calculus into the i-calculus

3.1 A typical call-by-value λ-calculus

In this section, we present a call-by-value λ-calculus with tuples, tagged variants, and integers. There are only two non-standard aspects of our treatment. First, for tuples and tagged variants, we distinguish between unevaluated forms ($(E_1, \ldots, E_n)$ and $\mathsf{in}_i\ E$) and evaluated forms ($[V_1, \ldots, V_n]$ and $\mathsf{IN}_i\ E$). Second, we explicitly identify the erroneous terms and have rewriting rules that transform them to the special form undef. Both of these aspects are only for convenience.

We define the syntax of our call-by-value λ-calculus as follows:

$$\begin{array}{lll}
y, z \in \mathsf{Variable} & & \\
V \in \mathsf{Value} & ::= & i \mid y \mid [V_1, \ldots, V_n] \mid \lambda y.\, E \mid \mathsf{IN}_i\ V \\
E \in \mathsf{Expression} & ::= & \mathsf{let}\ y = E_1\ \mathsf{in}\ E_2 \\
 & & \mid E_1\ E_2 \mid \pi_i\ E \mid V \mid \mathsf{in}_i\ E \mid (E_1, \ldots, E_n) \mid \mathsf{undef} \\
 & & \mid \mathsf{case}\ E\ \mathsf{of}\ \mathsf{IN}_1\ y \Longrightarrow E_1, \ldots, \mathsf{IN}_k\ y \Longrightarrow E_k \\
 & & \mid \mathsf{if}\ E_1 = E_2\ \mathsf{then}\ E_3\ \mathsf{else}\ E_4 \mid E_1 + E_2 \mid \ldots
\end{array}$$

We define the rewriting rules of our call-by-value λ-calculus in the following categories:

- Normal evaluation:

$$\begin{array}{lcll}
\mathsf{if}\ i = i'\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 & \longrightarrow & \begin{cases} E_1 \text{ if } i = i' \\ E_2 \text{ if } i \neq i' \end{cases} & \\
\mathsf{in}_i\ V & \longrightarrow & \mathsf{IN}_i\ V & \\
\mathsf{case}\ \mathsf{IN}_i\ V\ \mathsf{of}\ \ldots, \mathsf{IN}_i\ y \Longrightarrow E_i, \ldots & \longrightarrow & E_i[y := V] & \\
(\lambda y.\, E)V & \longrightarrow & E[y := V] & \\
(V_0, \ldots, V_n) & \longrightarrow & [V_0, \ldots, V_n] & \\
\pi_i\ [V_0, \ldots, V_n] & \longrightarrow & V_i \text{ if } 0 \leq i \leq n & \\
\mathsf{let}\ y = V\ \mathsf{in}\ E & \longrightarrow & E[y := V] & \\
i_1 + i_2 & \longrightarrow & i_3 & \text{where } i_3 = i_1 + i_2 \text{ (addition, not syntax)}
\end{array}$$

- Errors:

$\mathsf{undef}\ E$	$\longrightarrow$	undef	
$V\ \mathsf{undef}$	$\longrightarrow$	undef	
$V\ E$	$\longrightarrow$	undef	if V not of form $\lambda y.E$ or y
$\pi_i\, V$	$\longrightarrow$	undef	if V not of form $[V_1,\ldots,V_n]$ or y
$\pi_i\,\mathsf{undef}$	$\longrightarrow$	undef	
$\pi_i\,[V_1,\ldots,V_n]$	$\longrightarrow$	undef	if $i \notin \{1,\ldots,n\}$
$\mathsf{if}\ \mathsf{undef} = E_2\ \mathsf{then}\ E_3\ \mathsf{else}\ E_4$	$\longrightarrow$	undef	
$\mathsf{if}\ V = \mathsf{undef}\ \mathsf{then}\ E_3\ \mathsf{else}\ E_4$	$\longrightarrow$	undef	
$\mathsf{if}\ V_1 = V_2\ \mathsf{then}\ E_3\ \mathsf{else}\ E_4$	$\longrightarrow$	undef	if V_1 or V_2 of form $\lambda y.E$ or $[V_1,\ldots,V_n]$ or $\mathsf{IN}_i\, V$
$\mathsf{undef} + E$	$\longrightarrow$	undef	
$V_1 + \mathsf{undef}$	$\longrightarrow$	undef	
$V_1 + V_2$	$\longrightarrow$	undef	if V_1 or V_2 of form $\lambda y.E$ or $[V_1,\ldots,V_n]$ or $\mathsf{IN}_i\, V$
$\mathsf{case}\ V\ \mathsf{of}\ \ldots$	$\longrightarrow$	undef	if V not of form $\mathsf{IN}_i\, V'$ or y
$\mathsf{case}\ \mathsf{undef}\ \mathsf{of}\ \ldots$	$\longrightarrow$	undef	
$\mathsf{case}\ \mathsf{IN}_i\, V\ \mathsf{of}\ \mathsf{IN}_1\, x \Longrightarrow E_1, \ldots, \mathsf{IN}_k\, x \Longrightarrow E_k$	$\longrightarrow$	undef	if $i \notin \{1,\ldots,k\}$
$\mathsf{let}\ y = \mathsf{undef}\ \mathsf{in}\ E$	$\longrightarrow$	undef	
$\mathsf{in}_i\ \mathsf{undef}$	$\longrightarrow$	undef	
$(V_1,\ldots,V_n,\mathsf{undef},E_1,\ldots,E_m)$	$\longrightarrow$	undef	

3.2 The translation

We translate λ-terms into the i-calculus using a fairly normal continuation-passing style translation. Every λ-calculus variable y translates into an i-calculus static location $y.0$ for a base y with maximum offset 0. In addition, the translation uses two distinct dynamic locations, $a.0$ for function arguments, and $k.0$ for the current continuation. We assume the bases a and k have maximum offset 0, although they are left free so this is in fact not specified in the result of the translation.

WARNING: This translation is currently slightly broken, in the sense that directed rewriting steps of our λ-calculus are not perfectly preserved. We know how to fix it but have not yet done so, because our priority has been on proving correctness of the i-calculus.

The translation is as follows:

$$
\begin{array}{ll}
\mathsf{Tv}(\mathsf{let}\, y = E_1\, \mathsf{in}\, E_2) & = \{a := \mathsf{junk}\};\ \mathsf{alloc}\ \{k_0\};\ k_0 := !k; \\
 & \quad k := [\mathsf{alloc}\ \{y\};\ y := !a;\ k := !k_0;\ \mathsf{Tv}(E_2)];\ \mathsf{Tv}(E_1) \\
 & \quad (\text{where } k_0 \in \mathsf{StaticBase}) \\
\mathsf{Tv}(E_1\ E_2) & = \mathsf{Tv}(\mathsf{let}\, y = E_1\, \mathsf{in}\, \mathsf{let}\, z = E_2\, \mathsf{in}\, y\ z) \\
 & \quad \text{if } E_1 \text{ or } E_2 \notin \mathsf{Value} \\
\mathsf{Tv}(V_1\ V_2) & = \mathsf{alloc}\ \{\};\ \{a := \mathsf{Tv}'(V_2)\};\ \mathsf{jmp}\ \mathsf{Tv}'(V_1) \\
\mathsf{Tv}(\pi_i\, E) & = \mathsf{Tv}(\mathsf{let}\, y = E\, \mathsf{in}\, \pi_i\, y) \\
 & \quad \text{if } E \notin \mathsf{Value} \\
\mathsf{Tv}(\pi_i\, V) & = \mathsf{alloc}\ \{\};\ \{a := \mathsf{Tv}'(V)\};\ a \mathrel{+}= i;\ a := !(!a);\ \mathsf{jmp}\ !k \\
\mathsf{Tv}((E_1, \ldots, E_n)) & = \mathsf{Tv}(\mathsf{let}\, y_1 = E_1\, \mathsf{in} \ldots \mathsf{let}\, y_n = E_n\, \mathsf{in}\, [y_1, \ldots, y_n]) \\
\mathsf{Tv}(\mathsf{in}_i\, E) & = \mathsf{Tv}(\mathsf{let}\, y = E\, \mathsf{in}\, [i, y]) \\
\mathsf{Tv}(\mathsf{case}\, E\, \mathsf{of} \ldots, \mathsf{IN}_i\, z \Longrightarrow E_i, \ldots) & = \mathsf{Tv}(\mathsf{let}\, y = E\, \mathsf{in}\, \mathsf{case}\, y\, \mathsf{of} \ldots, \mathsf{IN}_i\, z \Longrightarrow E_i, \ldots) \\
 & \quad \text{if } E \notin \mathsf{value} \\
\mathsf{Tv}(\mathsf{case}\, V\, \mathsf{of} \ldots, \mathsf{IN}_i\, z \Longrightarrow E_i, \ldots) & = \mathsf{Tv}(\mathsf{let}\, y = \pi_0\, V \\
 & \qquad \mathsf{in}\, \mathsf{let}\, z = \pi_1\, V \\
 & \qquad\quad \mathsf{in}\, \mathsf{if}\, y = 0\, \mathsf{then}\, E_0\, \mathsf{else} \ldots \mathsf{if}\, y = n\, \mathsf{then}\, E_n\, \mathsf{else}\, \mathsf{undef}) \\
\mathsf{Tv}(\mathsf{if}\, E_1 = E_2\, \mathsf{then}\, E_3\, \mathsf{else}\, E_4) & = \mathsf{Tv}(\mathsf{let}\, y_1 = E_1\, \mathsf{in}\, \mathsf{let}\, y_2 = E_2\, \mathsf{in}\, \mathsf{if}\, y_1 = y_2\, \mathsf{then}\, E_3\, \mathsf{else}\, E_4) \\
 & \quad \text{if } E_1 \notin \mathsf{Value} \text{ or } E_2 \notin \mathsf{Value} \\
\mathsf{Tv}(\mathsf{if}\, V = V'\, \mathsf{then}\, E_1\, \mathsf{else}\, E_2) & = \mathsf{alloc}\ \{\};\ \{a := \mathsf{Tv}'(V)\}; \\
 & \quad a \mathrel{-}= \mathsf{Tv}'(V');\ \mathsf{bnz}\ !a\ [\mathsf{Tv}(E_2)];\ \mathsf{Tv}(E_1) \\
\mathsf{Tv}(E_1 + E_2) & = \mathsf{Tv}(\mathsf{let}\, y = E_1\, \mathsf{in}\, \mathsf{let}\, z = E_2\, \mathsf{in}\, y + z) \\
 & \quad \text{if } \{E_1, E_2\} \not\subset \mathsf{Value} \\
\mathsf{Tv}(V_1 + V_2) & = \mathsf{alloc}\ \{\};\ \{a := \mathsf{Tv}'(V_1)\};\ a \mathrel{+}= \mathsf{Tv}'(V_2);\ \mathsf{jmp}\ !k \\
\mathsf{Tv}(\mathsf{undef}) & = \mathsf{undef} \\
\mathsf{Tv}(V) & = \mathsf{alloc}\ \{\};\ \{a := \mathsf{Tv}'(V)\};\ \mathsf{jmp}\ !k \\
\mathsf{Tv}'(y) & = !y \\
\mathsf{Tv}'(i) & = i \\
\mathsf{Tv}'(\lambda y.\, E) & = [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)] \\
\mathsf{Tv}'([V_0, \ldots, V_n]) & = \mathsf{alloc}\ \{x(n)\};\ \{x.i := \mathsf{Tv}'(V_i) \mid 0 \le i \le n\};\ x.0 \\
\mathsf{Tv}'(\mathsf{IN}_i\, V) & = \mathsf{Tv}'([i, V])
\end{array}
$$

3.3 Verifying the translation is faithful

In this section, we look at each of the rewriting rules of our λ-calculus and verify that it is simulated on the results of the translation by i-calculus rewriting rules.

We have not yet written the simulation for the rewriting rule for addition, nor have we written the simulations for the error rules. We think these are straightforward.

WARNING: As mentioned already, this translation is slightly broken.

- $\mathsf{if}\ i = i'\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 \longrightarrow \begin{cases} E_1 & \text{if } i = i' \\ E_2 & \text{if } i \neq i' \end{cases}$

$$
\begin{array}{lll}
 & \mathsf{Tv}(\mathsf{if}\ i = i'\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2) & \\
= & \{a := \mathsf{Tv}'(i)\};\ a -= \mathsf{Tv}'(i');\ \mathsf{bnz}\ !a\ [\mathsf{Tv}(E_2)];\ \mathsf{Tv}(E_1) & \\
= & \{a := i\};\ a -= i';\ \mathsf{bnz}\ !a\ [\mathsf{Tv}(E_2)];\ \mathsf{Tv}(E_1) & \\
\xrightarrow{\underline{\mathrm{A}_1}} & \{a := i''\};\ \mathsf{bnz}\ !a\ [\mathsf{Tv}(E_2)];\ \mathsf{Tv}(E_1) & \text{where } i'' = i - i' \\
\xrightarrow{\underline{\mathrm{D}_3}} & \{a := i''\};\ \mathsf{bnz}\ i''\ [\mathsf{Tv}(E_2)];\ \mathsf{Tv}(E_1) & \\
\xrightarrow{\underline{\mathrm{C}_2}} & \{a := i''\};\ \mathsf{Tv}(E_2) & \text{if } i \neq i' \\
\xrightarrow{\underline{\mathrm{D}_2}} & \mathsf{Tv}(E_2) &
\end{array}
$$

- $\mathsf{let}\ y = V\ \mathsf{in}\ E \longrightarrow E[y := V]$

$$
\begin{array}{ll}
 & \mathsf{Tv}(\mathsf{let}\ y = V\ \mathsf{in}\ E) \\
= & \{a := \mathsf{junk}\};\ \mathsf{alloc}\ \{k_0\};\ k_0 := !k;\ k := [\mathsf{alloc}\ \{y\};\ y := !a;\ k := !k_0;\ \mathsf{Tv}(E)];\ \mathsf{Tv}(V) \\
= & \{a := \mathsf{junk}\};\ \mathsf{alloc}\ \{k_0\};\ k_0 := !k; \\
 & k := [\mathsf{alloc}\ \{y\};\ y := !a;\ k := !k_0;\ \mathsf{Tv}(E)];\ \{a := \mathsf{Tv}'(V)\};\ \mathsf{jmp}\ !k \\
\xrightarrow{\underline{\mathrm{S}_3}\,\underline{\mathrm{S}_5}\,\underline{\mathrm{S}_5}\,\underline{\mathrm{S}_2}\,\underline{\mathrm{S}_2}} & \\
\xrightarrow{\underline{\mathrm{S}_2}} & \mathsf{alloc}\ \{k_0\};\ \{a := \mathsf{Tv}'(V), k_0 := !k, k := [\mathsf{alloc}\ \{y\};\ y := !a;\ k := !k_0;\ \mathsf{Tv}(E)]\};\ \mathsf{jmp}\ !k \\
\xrightarrow{\underline{\mathrm{D}_3}\,\underline{\mathrm{C}_1}} & \mathsf{alloc}\ \{k_0\};\ \{a := \mathsf{Tv}'(V), k_0 := !k, k := [\mathsf{alloc}\ \{y\};\ y := !a;\ k := !k_0;\ \mathsf{Tv}(E)]\}; \\
 & \mathsf{alloc}\ \{y\};\ y := !a;\ k := !k_0;\ \mathsf{Tv}(E) \\
\xrightarrow{\underline{\mathrm{S}_5}\,\underline{\mathrm{S}_5}\,\underline{\mathrm{S}_3}\,\underline{\mathrm{S}_1}\,\underline{\mathrm{S}_2}\,\underline{\mathrm{S}_2}\,\underline{\mathrm{G}_2}\,\underline{\mathrm{G}_3}} & \\
\xrightarrow{\underline{\mathrm{G}_1}} & \mathsf{alloc}\ \{y\};\ \{a := \mathsf{Tv}'(V), y := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E) \\
\xrightarrow{\underline{\mathrm{T}}_{\twoheadrightarrow}} & \mathsf{alloc}\ \{y\};\ \{a := \mathsf{Tv}'(V), y := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E[y := V]) \\
\xrightarrow{\underline{\mathrm{G}_3}\,\underline{\mathrm{G}_1}} & \{a := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E[y := V]) \\
\xrightarrow{\underline{\mathrm{S}_2}} & \mathsf{Tv}(E[y := V])
\end{array}
$$

- $\mathsf{in}_i\ V \longrightarrow \mathsf{IN}_i\ V$

$$
\begin{array}{ll}
 & \mathsf{Tv}(\mathsf{in}_i\ V) \\
= & \mathsf{Tv}(\mathsf{let}\ y = V\ \mathsf{in}\ [i, y]) \\
\longrightarrow\!\!\!\!\!\rightarrow & \mathsf{Tv}([i, V]) \\
= & \{a := \mathsf{Tv}'([i, V])\};\ \mathsf{jmp}\ !k \\
= & \{a := \mathsf{Tv}'(\mathsf{IN}_i\ V)\};\ \mathsf{jmp}\ !k \\
= & \mathsf{Tv}(\mathsf{IN}_i\ V)
\end{array}
$$

- $(\lambda y.\, E)\, V \longrightarrow E[y := V]$

$$
\begin{array}{ll}
 & \mathsf{Tv}((\lambda y.\, E)\, V) \\
= & \{a := \mathsf{Tv}'(V)\};\ \mathsf{jmp}\ \mathsf{Tv}'(\lambda y.\, E) \\
= & \{a := \mathsf{Tv}'(V)\};\ \mathsf{jmp}\ [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)] \\
\xrightarrow{C_1} & \{a := \mathsf{Tv}'(V)\};\ \mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E) \\
\xrightarrow{S_3}\xrightarrow{S_5}\xrightarrow{S_2} & \mathsf{alloc}\ \{y\};\ \{a := \mathsf{Tv}'(V), y := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E) \\
\xrightarrow{T}{}^{*} & \mathsf{alloc}\ \{y\};\ \{a := \mathsf{Tv}'(V), y := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E)[y := V] \\
\xrightarrow{G_3}\xrightarrow{G_1} & \{a := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E[y := V]) \\
\xrightarrow{S_2} & \mathsf{Tv}(E[y := V])
\end{array}
$$

- $(V_0, \ldots, V_n) \longrightarrow [V_0, \ldots, V_n]$

$$
\begin{array}{ll}
 & \mathsf{Tv}((V_0, \ldots, V_n)) \\
= & \mathsf{Tv}(\mathsf{let}\ y_0 = V_0\ \mathsf{in}\ \ldots \mathsf{let}\ y_n = V_n\ \mathsf{in}\ [y_0, \ldots, y_n]) \\
\longrightarrow{}^{*} & \mathsf{Tv}([V_0, \ldots, V_n])
\end{array}
$$

- $\pi_i\, [V_0, \ldots, V_n] \longrightarrow V_i$ where $0 \le i \le n$

$$
\begin{array}{ll}
 & \mathsf{Tv}(\pi_i\, [V_0, \ldots, V_n]) \\
= & \{a := \mathsf{Tv}'([V_0, \ldots, V_n])\};\ a \mathrel{+}= i;\ a := !(!a);\ \mathsf{jmp}\ !k \\
= & \{a := (\mathsf{alloc}\ \{x(n)\};\ \{x.i := \mathsf{Tv}'(V_i) \mid 0 \le i \le n\};\ x.0)\}; \\
 & a \mathrel{+}= i;\ a := !(!a);\ \mathsf{jmp}\ !k \\
\xrightarrow{S_1} & \mathsf{alloc}\ \{x(n)\};\ \{x.i := \mathsf{Tv}'(V_i) \mid 0 \le i \le n\}; \\
 & \{a := x.0\};\ a \mathrel{+}= i;\ a := !(!a);\ \mathsf{jmp}\ !k \\
= & \mathsf{alloc}\ \{x(n)\};\ \{x.0 := \mathsf{Tv}'(V_0), \ldots, x.n := \mathsf{Tv}'(V_n)\}; \\
 & \{a := x.0\};\ a \mathrel{+}= i;\ a := !(!a);\ \mathsf{jmp}\ !k \\
 & \\
\xrightarrow{S_2} & \mathsf{alloc}\ \{x(n)\};\ \{x.0 := \mathsf{Tv}'(V_0), \ldots, x.n := \mathsf{Tv}'(V_n), a := x.0\}; \\
 & a \mathrel{+}= i;\ a := !(!a);\ \mathsf{jmp}\ !k \\
\xrightarrow{A_2} & \mathsf{alloc}\ \{x(n)\};\ \{x.0 := \mathsf{Tv}'(V_0), \ldots, x.n := \mathsf{Tv}'(V_n), a := x.i\}; \\
 & a := !(!a);\ \mathsf{jmp}\ !k \\
\xrightarrow{D_4} & \mathsf{alloc}\ \{x(n)\};\ \{x.0 := \mathsf{Tv}'(V_0), \ldots, x.n := \mathsf{Tv}'(V_n), a := x.i\}; \\
 & a := !(x.i);\ \mathsf{jmp}\ !k \\
\xrightarrow{S_5}\xrightarrow{S_2} & \mathsf{alloc}\ \{x(n)\}; \\
 & \{x.0 := \mathsf{Tv}'(V_0), \ldots, x.n := \mathsf{Tv}'(V_n), a := \mathsf{Tv}'(V_i)\};\ \mathsf{jmp}\ !k \\
\xrightarrow{G_3}\xrightarrow{G_1} & \{a := \mathsf{Tv}'(V_i)\};\ \mathsf{jmp}\ !k \\
= & \mathsf{Tv}(V_i)
\end{array}
$$

- $\mathsf{case}\ \mathsf{IN}_i\ V\ \mathsf{of}\ \ldots, \mathsf{IN}_i\ z \Longrightarrow E_i, \ldots \longrightarrow E_i[z := V]$

  $\mathsf{Tv}(\mathsf{case}\ \mathsf{IN}_i\ V\ \mathsf{of}\ \ldots, \mathsf{IN}_i\ z \Longrightarrow E_i, \ldots)$

  $=$ $\mathsf{Tv}(\mathsf{let}\ y = \pi_0\ \mathsf{IN}_i\ V$
  $\quad \mathsf{in\ let}\ z = \pi_1\ \mathsf{IN}_i\ V\ \mathsf{in\ if}\ y = 0\ \mathsf{then}\ E_0\ \mathsf{else}\ \ldots \mathsf{if}\ y = n\ \mathsf{then}\ E_n\ \mathsf{else\ undef})$

  $=$ $\mathsf{Tv}(\mathsf{let}\ y = \pi_0\ [i, V]$
  $\quad \mathsf{in\ let}\ z = \pi_1\ [i, V]\ \mathsf{in\ if}\ y = 0\ \mathsf{then}\ E_0\ \mathsf{else}\ \ldots \mathsf{if}\ y = n\ \mathsf{then}\ E_n\ \mathsf{else\ undef})$

  $\twoheadrightarrow$ $\mathsf{Tv}(\mathsf{if}\ i = 0\ \mathsf{then}\ E_0[z := V]\ \mathsf{else}\ \ldots \mathsf{if}\ i = n\ \mathsf{then}\ E_n[z := V]\ \mathsf{else\ undef})$

  $\twoheadrightarrow$ $\mathsf{Tv}(E_i[z := V])$

3.4 Alternative definition of the translation

In this section, we give an alternative definition of Tv that avoids converting to A-normal forms first. We only give the changed case for applications:

$\mathsf{Tv}(E_1\ E_2)$

$=$ $\{a := \mathsf{junk}\};\ \mathsf{alloc}\ \{k_0, a_1\};\ k_0 := !k;\ k := [a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ \mathsf{Tv}(E_2)];\ \mathsf{Tv}(E_1)$

The verification that the β-reduction rule is preserved by the translation is as follows.

WARNING: This translation shares the same problem as the earlier translation (this shows up as "missing rule" annotations below). We know how to fix it.

WARNING: There is a small typesetting error below where "$\mathsf{jmp}\ !k$" wrongly jumps up to the end of the preceding line from where it should be. This is some LaTeX macro madness that we will fix soon.

$$
\begin{array}{ll}
 & \mathsf{Tv}((\lambda y.\,E)\,V) \\
= & \{a := \mathsf{junk}\};\ \mathsf{alloc}\ \{k_0, a_1\};\ k_0 := !k; \\
 & k := [a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ \mathsf{Tv}(V)];\ \mathsf{Tv}(\lambda y.\,E) \\
= & \{a := \mathsf{junk}\};\ \mathsf{alloc}\ \{k_0, a_1\};\ k_0 := !k; \\
 & k := [a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ a := \mathsf{Tv}'(V);\ \mathsf{jmp}\ !k]; \\
 & \{a := \mathsf{Tv}'(\lambda y.\,E)\};\ \mathsf{jmp}\ !k \\
= & \{a := \mathsf{junk}\};\ \mathsf{alloc}\ \{k_0, a_1\};\ k_0 := !k; \\
 & k := [a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ a := \mathsf{Tv}'(V);\ \mathsf{jmp}\ !k]; \\
 & \{a := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)]\};\ \mathsf{jmp}\ !k \\
\underline{\mathrm{S}_3}\,\underline{\mathrm{S}_4}\,\underline{\mathrm{S}_4}\,\underline{\mathrm{S}_2}\,\underline{\mathrm{S}_2} & \\
\underline{\mathrm{S}_2} & \mathsf{alloc}\ \{k_0, a_1\}; \\
 & \{k_0 := !k, k := [a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ a := \mathsf{Tv}'(V);\ \mathsf{jmp}\ !k], \mathsf{jmp}\ !k \\
 & \ a := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)]\}; \\
\underline{\mathrm{D}_4}\,\underline{\mathrm{C}_1} & \mathsf{alloc}\ \{k_0, a_1\}; \\
 & \{k_0 := !k, k := [a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ a := \mathsf{Tv}'(V);\ \mathsf{jmp}\ !k], \\
 & \ a := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)]\}; \\
 & a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ a := \mathsf{Tv}'(V);\ \mathsf{jmp}\ !k \\
\underline{\mathrm{S}_4}\,\underline{\mathrm{S}_4}\,\underline{\mathrm{S}_4}\,\underline{\mathrm{S}_2} & \\
\underline{\mathrm{S}_2} & \mathsf{alloc}\ \{k_0, a_1\}; \\
 & \{k_0 := !k, k := [a_1 := !a;\ k := [k := !k_0;\ \mathsf{jmp}\ !a_1];\ a := \mathsf{Tv}'(V);\ \mathsf{jmp}\ !k], \\
 & \ a := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)]\}; \\
 & \{a_1 := !a, k := [k := !k_0;\ \mathsf{jmp}\ !a_1], a := \mathsf{Tv}'(V)\};\ \mathsf{jmp}\ !k \\
\underline{\mathrm{S}_2} & \mathsf{alloc}\ \{k_0, a_1\};\ \{k_0 := !k, a_1 := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)], \mathsf{jmp}\ !k \\
 & \qquad\qquad k := [k := !k_0;\ \mathsf{jmp}\ !a_1], a := \mathsf{Tv}'(V)\}; \\
\underline{\mathrm{D}_4}\,\underline{\mathrm{C}_1} & \mathsf{alloc}\ \{k_0, a_1\};\ \{k_0 := !k, a_1 := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)], k := !k_0;\ \mathsf{jmp}\ !a_1 \\
 & \qquad\qquad k := [k := !k_0;\ \mathsf{jmp}\ !a_1], a := \mathsf{Tv}'(V)\}; \\
\underline{\mathrm{S}_4}\,\underline{\mathrm{S}_2} & \mathsf{alloc}\ \{k_0, a_1\}; \\
 & \{k_0 := !k, a_1 := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)], a := \mathsf{Tv}'(V), k := !k\};\ \mathsf{jmp}\ !a_1 \\
\underline{\mathrm{D}_4}\,\underline{\mathrm{C}_1} & \mathsf{alloc}\ \{k_0, a_1\};\ \{k_0 := !k, a_1 := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)], a := \mathsf{Tv}'(V), k := !k\}; \\
 & \mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E) \\
\underline{\mathrm{S}_3}\,\underline{\mathrm{S}_1}\,\underline{\mathrm{S}_4} & \\
\underline{\mathrm{S}_2} & \mathsf{alloc}\ \{k_0, a_1, y\}; \\
 & \{k_0 := !k, a_1 := [\mathsf{alloc}\ \{y\};\ y := !a;\ \mathsf{Tv}(E)], a := \mathsf{Tv}'(V), k := !k, y := \mathsf{Tv}'(V)\}; \\
 & \mathsf{Tv}(E) \\
\underline{\mathrm{G}_3}\,\underline{\mathrm{G}_1} & \mathsf{alloc}\ \{y\};\ \{a := \mathsf{Tv}'(V), k := !k, y := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E) \\
\underline{\mathrm{G}_2} & \mathsf{alloc}\ \{y\};\ \{a := \mathsf{Tv}'(V), y := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E) \\
\underline{\mathrm{T}_{\gg}} & \mathsf{alloc}\ \{y\};\ \{a := \mathsf{Tv}'(V), y := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E[y := V]) \\
\underline{\mathrm{G}_3}\,\underline{\mathrm{G}_1} & \mathsf{alloc}\ \{\};\ \{a := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E[y := V]) \\
\xrightarrow{\text{missing rule}} & \{a := \mathsf{Tv}'(V)\};\ \mathsf{Tv}(E[y := V]) \\
\underline{\mathrm{S}_1}\,\underline{\mathrm{S}_4}_{\gg}\,\underline{\mathrm{S}_2}_{\gg}\,\underline{\mathrm{G}_3}\,\underline{\mathrm{G}_1} & \\
\xrightarrow{\text{missing rule}} & \mathsf{Tv}(E[y := V])
\end{array}
$$

4 The AES framework

We will explain here the structures and methods we are going to use in our proof. The following definitions are taken from [1].

4.1 Mathematical definitions

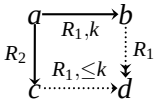
Let R range over binary relations (only within this subsection). Let $\xrightarrow{R}$ and $\overset{R}{\rightarrow}$ be alternate notations for R which are usable infix, i.e., both $a \xrightarrow{R} b$ and $a \overset{R}{\rightarrow} b$ stand for $R(a,b)$ which in turn stands for $(a,b) \in R$.

Define the following operators on binary relations. Let $R;R'$ be the composition of R with R' (i.e., $\{\,(a,b)\,|\,\exists c.\, R(a,c) \text{ and } R'(c,b)\,\}$). Let $\xrightarrow{R,0} = R^0$ be equality at the type intended for R. Let $\xrightarrow{R,i+1} = R^{i+1} = (R^i;R)$ when $0 \le i$. Let $\xrightarrow{R,\ge k} = R^{\ge k} = \bigcup_{i \ge k} R^i$. Let $\xrightarrow{R,\le k} = R^{\le k} = \bigcup_{i \le k} R^i$. Let $\xrightarrow{R,j,\le k} = R^j \cap R^{\le k}$ (useful in diagrams when j is existentially quantified). Let $\xrightarrow{R}\!\!\!\!\!\rightarrow = R^* = R^{\ge 0}$ (the transitive, reflexive closure).

Let $\xleftarrow{R} = R^{-1}$ be the inverse of R (i.e., $\{\,(a,b)\,|\,R(b,a)\,\}$). Let $\xleftarrow[R]{}\!\!\!\!\!\leftarrow = (R^{-1})^{\ge 0}$. Let $\xleftrightarrow{R} = (R \cup R^{-1})$ (the symmetric closure). When $R = R^{-1}$ (i.e., R is symmetric), let $\xleftrightarrow{R} = R$. Let $\xleftrightarrow{R,k} = (\xleftrightarrow{R})^k$. Let $\xleftarrow[R]{}\!\!\!\!\!\leftarrow = (\xleftrightarrow{R})^{\ge 0}$.

Let an entity a *be* a R*-normal form*, written $\mathsf{is\text{-}nf}(R,a)$, iff there does not exist some entity b such that $a \xrightarrow{R} b$. Let an entity a *have* a R-normal form, written $\mathsf{has\text{-}nf}(R,a)$, iff there exists some b such that $a \xrightarrow{R}\!\!\!\!\!\rightarrow b$ and $\mathsf{is\text{-}nf}(R,b)$. Let $\xrightarrow{R,\mathsf{nf}}\!\!\!\!\!\rightarrow$ be the relation such that $a \xrightarrow{R,\mathsf{nf}}\!\!\!\!\!\rightarrow b$ iff $a \xrightarrow{R}\!\!\!\!\!\rightarrow b$ and $\mathsf{is\text{-}nf}(R,b)$. A relation R is *terminating* (a.k.a. *strongly normalizing*), written $\mathsf{Trm}(R)$, iff there does not exist any total function f with as domain the natural numbers such that $R(f(i), f(i+1))$ for all $i \ge 0$. A relation R is *(weakly) normalizing*, written $\mathsf{Nrm}(R)$, iff for every entity a there is some entity b such that $a \xrightarrow{R,\mathsf{nf}}\!\!\!\!\!\rightarrow b$.

Diagrams make statements about relations where solid and dotted edges indicate quantification. Metavariables already mentioned outside the diagram are unquantified. Other metavariables (e.g., for node names or used in edge labels) are universally quantified if attached to a solid edge and existentially quantified if attached only to dotted edges. As an example, in a context where R_1 and R_2 have already been given, the following equivalence holds:



$$\iff \forall a,b,c,k.\ (a \xrightarrow{R_1,k} b \wedge a \xrightarrow{R_2} c) \Rightarrow \exists d.\ c \xrightarrow{R_1,\le k} d \wedge b \xrightarrow{R_1}\!\!\!\!\!\rightarrow d$$

In proofs, the reason for each diagram polygon will usually be written inside it.

In all cases the symbol "=" indicates mathematical equality (as it always should).

4.2 Abstract evaluation systems

An *abstract evaluation system* (AES) is a tuple

$$(\mathbb{T}, \mathbb{S}, \mathbb{R}, \mathsf{endpoints}, \mathbb{E}, \mathsf{result})$$

satisfying the conditions given below by axioms 4.2 and 4.3 and the immediately following conditions. The carriers of an AES are the sets $\mathbb{T}$, $\mathbb{S}$, and $\mathbb{R}$. The function endpoints maps $\mathbb{S}$ to $\mathbb{T} \times \mathbb{T}$. The set $\mathbb{E}$ is a subset of $\mathbb{S}$. The function result maps $\mathbb{T}$ to $\mathbb{R}$. Let t range over $\mathbb{T}$, let s range over $\mathbb{S}$, let r range over $\mathbb{R}$, and let $\mathbb{S}$ range over subsets of $\mathbb{S}$.

The intended meaning is as follows. $\mathbb{T}$ should be a set of terms. $\mathbb{S}$ should be a set of rewriting steps. $\mathbb{R}$ should be a set of evaluation results which by axiom 4.3(1) will most likely

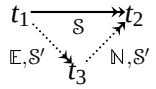
contain the symbol diverges and one or more other members, typically symbols such as halt, error, etc. The halt case might be subdivided into possible constant values of final results. In examples, if $\mathbb{R}$ is not specified then assume this means $\mathbb{R} = \{\mathsf{diverges}, \mathsf{halt}\}$. If $\mathsf{endpoints}(s) = (t_1, t_2)$, this should mean that step s rewrites term t_1 to term t_2. The members of $\mathbb{E}$ are the rewriting steps used for evaluation. Let $\mathbb{N} = \mathbb{S} \setminus \mathbb{E}$ (where "$\mathbb{N}$" stands for "non-evaluation"). If $\mathsf{result}(t) = r$, this should mean that r is the observable result of evaluating term t, where diverges is reserved by axiom 4.3(1) for non-halting evaluations.

Let rewriting notation be defined as follows. Given a rewriting step set $\mathbb{S}$, let $\rightarrow_{\mathbb{S}}$ be the binary relation $\{ (t,t') \mid \exists s \in \mathbb{S}.\ \mathsf{endpoints}(s) = (t,t') \}$. Thus, $t \xrightarrow{\mathbb{S}} t'$ iff there exists $s \in \mathbb{S}$ such that $\mathsf{endpoints}(s) = (t,t')$. When a rewriting step set $\mathbb{S}$ is used in a context *requiring* a binary relation on $\mathbb{T}$, then let $\mathbb{S}$ implicitly stand for $\rightarrow_{\mathbb{S}}$. Thus, as examples, $t \xrightarrow{\mathbb{S}} t'$ stands for $t \xrightarrow{\mathbb{S}} t'$ and an $\mathbb{S}$-normal form is simply a $\rightarrow_{\mathbb{S}}$-normal form. When used in a position *requiring* a subset of $\mathbb{S}$ or a binary relation on $\mathbb{T}$, let s stand for $\{s\}$ and let $\mathbb{S}, \mathbb{S}'$ stand for $\mathbb{S} \cap \mathbb{S}'$. Thus, as an example, $t \xrightarrow{\mathbb{S},s} t'$ stands for $t \xrightarrow{\mathbb{S} \cap \{s\}} t'$. When a binary relation on $\mathbb{T}$ is *required* and none is supplied, then let the relation $\rightarrow_{\mathbb{S}}$ be implicitly supplied. Thus, as examples, $t \longrightarrow t'$ stands for $t \xrightarrow{\mathbb{S}} t'$ and $t \xrightarrow{k} t'$ stands for $t \xrightarrow{\mathbb{S},k} t'$.

Definition 4.1 (Rewriting Step Set Properties). Define the following rewrite step sets and properties of rewriting step sets:

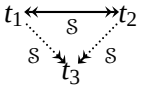
Standardization:

$\mathsf{Std}(\mathbb{S}, \mathbb{S}') \iff$



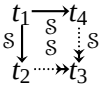
Confluence:

$\mathsf{Conf}(\mathbb{S}) \iff$



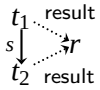
Local Confluence:

$\mathsf{LConf}(\mathbb{S}) \iff$



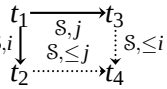
Meaning Preservation:

$s \in \mathsf{MP} \iff$



Subcommutativity:

$\mathsf{SubComm}(\mathbb{S}, i, j) \iff$



Let $\mathsf{Std}(\mathbb{S})$ abbreviate $\mathsf{Std}(\mathbb{S}, \mathbb{S})$. Let $\mathsf{SubComm}(\mathbb{S})$ abbreviate $\mathsf{SubComm}(\mathbb{S}, 1, 1)$. Traditionally, only $\mathsf{Std}(\mathbb{S}) = \mathsf{Std}(\mathbb{S}, \mathbb{S})$ is considered. The simple definition of MP is reasonable because axiom 4.3(1) (given below) means MP implies preservation of the existence of $\mathbb{E}$-normal forms. See also warning **??** and convention **??** and do not confuse MP with observational equivalence. □

Axiom 4.2 (Subcommutativity of Evaluation). $\mathsf{SubComm}(\mathbb{E})$. □

Axiom 4.3 (Evaluation Sanity).

1. **"diverges" Means Evaluation Diverges**:
   $\mathsf{result}(t) = \mathsf{diverges} \Leftrightarrow \neg\mathsf{has\text{-}nf}(\mathbb{E}, t)$.

2. **Evaluation Steps Preserve Meaning**:
   $\mathbb{E} \subseteq \mathsf{MP}$.

3. **Non-Evaluation Steps Preserve Evaluation Steps**:

Consequently, if $t_1 \xrightarrow{\mathbb{N}} t_2$, then $\mathsf{is\text{-}nf}(\mathbb{E}, t_1) \Leftrightarrow \mathsf{is\text{-}nf}(\mathbb{E}, t_2)$.

4. **Non-Evaluation Steps on $\mathbb{E}$-Normal Forms Preserve Meaning**:
If $t \xrightarrow{\mathbb{N}} t'$ and $\mathsf{is\text{-}nf}(\mathbb{E}, t)$, then $\mathsf{result}(t) = \mathsf{result}(t')$. □

4.3 Elementary diagrams for meaning preservation

This section provides abstract methods for proving properties for particular rewriting step sets. Definition 4.4 defines a property called WB\Std which will enable us to prove as shown in the theorem 4.5 that a set of steps is meaning preserving (see [1] for the proof).

Definition 4.4 (Well Behaved Rewriting Step Sets). Define the following rewriting step set properties:

Weak Lift/Project 1-Step:

$$\mathsf{WLP1}(\mathcal{S}) \iff \begin{array}{ccc} t_1 & \longrightarrow & t_4 \\ {\scriptstyle \mathbb{N},\mathcal{S}}\uparrow & \mathbb{E} & \uparrow\downarrow \mathcal{S} \\ t_2 & \cdots\!\!\rightarrow & t_3 \end{array}$$

$\mathbb{N}$-Steps Do Not Create $\mathbb{E}$-Steps:

$$\mathsf{NE}(\mathcal{S}) \iff \begin{array}{ccc} t_1 & \cdots\!\!\rightarrow & t_4 \\ {\scriptstyle \mathbb{N},\mathcal{S}}\downarrow & \mathbb{E},\mathcal{S} & \\ t_2 & \longrightarrow & t_3 \end{array}$$

Well Behaved without Standardization:

$$\mathsf{WB\backslash Std}(\mathcal{S}) \iff \mathsf{Trm}(\mathcal{S}) \wedge \mathsf{LConf}(\mathcal{S}) \wedge \mathsf{WLP1}(\mathcal{S}) \wedge \mathsf{NE}(\mathcal{S})$$

□

Theorem 4.5 (Well Behaved Rewriting Step Sets). *If* $\mathsf{WB\backslash Std}(\mathcal{S})$, *then* $\mathcal{S} \subset \mathsf{MP}$.

□

5 Evaluation and correctness for the i-calculus

We will now define evaluation for the i-calculus and simultaneously express it as an instance of the AES framework. Then, we will use the tools of the AES framework to prove that the i-calculus is correct, i.e., all of its steps (both evaluation and non-evaluation) are meaning preserving.

This means first proving the i-calculus satisfies the basic AES axioms, and then proving $\mathsf{WB\backslash Std}(S)$ for each member S of a covering $\{S_i | 1 \leq i \leq n\}$ of the rewriting steps. The precise allocation of steps to members of the covering that we choose will depend on the diagrams we will have in section 5.5, and as this particular section is still incomplete, the covering itself is not yet decided.

Once we are done, because it also holds that the i-calculus rewriting relation is contextually closed, this will imply that i-calculus rewriting is contained in the usual notion of observational equivalence.

5.1 Terms and steps

The first AES components we define for the i-calculus are the sets of terms $\mathbb{T}$ and rewriting steps $\mathbb{S}$, and the rewriting step endpoint function $\mathsf{endpoints}$.

$$
\begin{array}{llcl}
 & \mathbb{T} & = & \mathsf{Program} \\
\\
\widetilde{P} & \in \mathsf{RewritingCtxt} & ::= & \widetilde{O};P \mid O;\widetilde{P} \mid \Box \\
\widetilde{O} & & ::= & \widetilde{S} \mid \widetilde{J} \\
\widetilde{B} & & ::= & V:=\widetilde{U} \mid \widetilde{T} \\
\widetilde{S} & & ::= & V\,A{=}\widetilde{U} \mid V:=!(\widetilde{U}) \mid \widetilde{B} \\
\widetilde{J} & & ::= & \mathsf{jmp}\,\widetilde{U} \mid \mathsf{bnz}\,\widetilde{U}\,W \mid \mathsf{bnz}\,W\,\widetilde{U} \\
\widetilde{T} & & ::= & \{\ell_1:=W_1,\ldots,\ell_k:=\widetilde{U_k},\ldots,\ell_n:=W_n\} \\
\widetilde{V} & & ::= & [\widetilde{P}] \\
\widetilde{U} & & ::= & \widetilde{V} \mid B;\widetilde{U} \mid \widetilde{B};U \\
\\
R & \in \mathsf{Rules} & = & \{\mathrm{G}_{1,X} \mid X \in \mathsf{Alloc}\} \cup \{\mathrm{G}_{3,T} \mid T \in \mathsf{StoreTransformer}\} \\
 & & & \{\mathrm{S}_i \mid 1 \le i \le 6\} \cup \{\mathrm{S}_{7,\ell} \mid \ell \in \mathsf{Dom}(T)\} \cup \\
 & & & \{\mathrm{T}_{\widehat{P}} \mid \widehat{P} \in \mathsf{SubstitutionCtxt}\} \cup \{\mathrm{D}_{1,\ell} \mid \ell \in \mathsf{Location}\} \cup \\
 & & & \{\mathrm{D}_2,\mathrm{D}_3,\mathrm{D}_4,\mathrm{C}_1,\mathrm{C}_2,\mathrm{C}_3,\mathrm{A}_1,\mathrm{A}_2\} \cup \{\mathrm{E}_i \mid 1 \le i \le 14\} \\
 & \mathsf{ErrorRule} & = & \{\mathrm{E}_i \mid 1 \le i \le 14\} \\
 & \mathsf{ErrorProg} & = & \{P \mid R \in \mathsf{ErrorRule} \wedge R(P) \text{ is defined}\} \\
\\
s & \in \mathbb{S} & = & \left\{ (P,\widetilde{P},R) \middle| \begin{array}{l} \widetilde{P} \in \mathsf{RewritingCtxt} \wedge \\ (P \in \mathsf{ErrorProg} \Longrightarrow R \in \mathsf{ErrorRule}) \wedge R(P) \text{ defined} \end{array} \right\} \\
 & \mathsf{endpoints}(P,\widetilde{P},R) & = & (\widetilde{P}[P],\widetilde{P}[R(P)])
\end{array}
$$

5.2 Evaluation and results

Now we define the other AES components for the i-calculus, namely the evaluation step set $\mathbb{E}$, the set of evaluation results $\mathbb{R}$, and the term result function result,

We begin by defining the set $\mathbb{E}'$ of *candidate* evaluation steps, which we will subsequently filter to make evaluation deterministic.

$$
\begin{array}{lcl}
\mathsf{EvalCtxt} & ::= & \Box \mid \mathsf{alloc}\,X;\Box \mid T;\Box \mid \mathsf{alloc}\,X;T;\Box \\
\\
X_1 & = & \{(\Box,R) \mid R \in \{\mathrm{S}_1,\mathrm{D}_2,\mathrm{D}_3,\mathrm{E}_6,\mathrm{E}_7,\mathrm{E}_8,\mathrm{E}_{12},\mathrm{E}_{13},\mathrm{E}_{14}\}\} \\
X_2 & = & \left\{ (\widetilde{P},R) \middle| \begin{array}{l} \widetilde{P} \in \{\Box,(\mathsf{alloc}\,X;\Box) \mid X \in \mathsf{Alloc}\} \wedge \\ R \in \{\mathrm{S}_2,\mathrm{S}_3,\mathrm{S}_{7,\ell},\mathrm{D}_4,\mathrm{A}_1,\mathrm{A}_2,\mathrm{E}_{11} \mid \ell \in \mathsf{Location}\} \end{array} \right\} \\
X_3 & = & \left\{ (\widetilde{P},R) \middle| \begin{array}{l} \widetilde{P} \in \mathsf{EvalCtxt} \wedge \\ R \in \{\mathrm{S}_4,\mathrm{S}_5,\mathrm{S}_6,\mathrm{C}_1,\mathrm{C}_2,\mathrm{C}_3,\mathrm{E}_1,\mathrm{E}_2,\mathrm{E}_3,\mathrm{E}_4,\mathrm{E}_5,\mathrm{E}_9,\mathrm{E}_{10}\} \end{array} \right\} \\
\mathbb{E}' & = & \{(P,\widetilde{P},R) \in \mathbb{S} \mid (\widetilde{P},R) \in X_1 \cup X_2 \cup X_3\}
\end{array}
$$

In order to make the evaluation steps $\mathbb{E}$ deterministic for the rules that are not error-rules, we will write here an algorithm in order to pick up only one evaluation step at a time. $\mathbb{E}$ will be built as a subset of $\mathbb{E}'$ with the following conventions. We must consider for each program P the set $\mathbb{E}_P = \{(P',\widetilde{P},R) \in \mathbb{E}' \,|\, \widetilde{P}[P'] = P\}$. If $\mathbb{E}_P$ has two or more members, then we must choose one.

- If $\mathbb{E}_P$ is a singleton, then let $\mathbb{E}_P \subset \mathbb{E}$.
- Otherwise, if $S = \{(P',\widetilde{P},R) \in \mathbb{E}_P \,|\, R \in \mathsf{ErrorRule}\} \neq \varnothing$, i.e., there are one or more steps using error rules, then we define $S' \subseteq S$ such that the steps in S' are those with the smallest

reduction context, with the following order : $\Box < \mathsf{alloc}\, X; \Box < T; \Box < \mathsf{alloc}\, X; T; \Box$. Consequently all the steps in S' will have the same reduction context $\widetilde{P}$. We then let $S' \subset \mathbb{E}$ and we do not care if it is not deterministic as it will be shown later.

- Otherwise, if $S = \{ (P', \widetilde{P}, R) \in \mathbb{E}_P \,|\, R = \mathrm{S}_{7,\ell} \text{ for some } \ell \} \neq \varnothing$, i.e., there are one or more steps using rule $\mathrm{S}_{7,\ell}$, then we pick a step as follows. Because of the way $\mathbb{E}'$ is defined, all of the steps in S must work on the same transformer in the same position. Specifically, it must hold that $S = \{ (P', \widetilde{P}, \mathrm{S}_{7,\ell_i}) \,|\, 1 \leq i \leq n \}$ for some fixed P', $\widetilde{P}$, and $n \geq 1$, and it must further hold that $P' = \{\ell'_1 := W_1, \ldots, \ell'_m := W_m\}; P''$ for some P''. We choose the member of S operating leftmost; specifically pick $j \in \{1, \ldots, m\}$ and $k \in \{1, \ldots, n\}$ so j is the smallest value such that $\ell'_j = \ell_k$, and then let $\mathrm{S}_{7,\ell_k} \in \mathbb{E}$.

- Otherwise, if $S = \{ (P', \widetilde{P}, R) \in \mathbb{E}_P \,|\, R \in \{\mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4, \mathrm{S}_5, \mathrm{C}_1, \mathrm{C}_2, \mathrm{C}_3, \mathrm{A}_1, \mathrm{A}_2\} \} \neq \varnothing$, then S must have exactly one member and we let $S \subset \mathbb{E}$.

- Otherwise, $\mathbb{E}_P$ must be of size one and mention only rules in the set $\{\mathrm{S}_6, \mathrm{D}_2, \mathrm{D}_3, \mathrm{D}_4\}$, and we let $\mathbb{E}_P \subset \mathbb{E}$.

Now we define results, thereby completing the AES definition for the i-calculus:

$$
\begin{array}{rcl}
r \quad \in \mathbb{R} & = & \{\mathsf{diverges}, \mathsf{halt}, \mathsf{wrong}\} \\
\mathsf{result}(P) & = & \begin{cases} \mathsf{diverges} & \text{if } \neg\text{has-nf}(\mathbb{E}, P) \\ \mathsf{halt} & \text{if } P \xrightarrow{\mathbb{E},\mathsf{nf}}\!\!\!\!\rightarrow\, P' \neq \mathsf{undef} \\ \mathsf{undef} & \text{if } P \xrightarrow{\mathbb{E},\mathsf{nf}}\!\!\!\!\rightarrow\, \mathsf{undef} \end{cases}
\end{array}
$$

5.3 Subcommutativity of evaluation

We now begin to prove the properties of the i-calculus as an AES. First, we explain why it satisfies axiom 4.2.

Consider a program P_0 such that there exists $s_1, s_2 \in \mathbb{E}$ where $P_0 \xrightarrow{s_i} P_i$ for $i \in \{1,2\}$. We need to show that there exists P_3 such that $P_i \xrightarrow{\mathbb{E},\leq 1} P_3$ for $i \in \{1,2\}$. If $s_1 = s_2$, this follows trivially by letting $P_3 = P_2 = P_1$. If $s_1 \neq s_2$, by the definition of $\mathbb{E}$, both s_1 and s_2 must be error steps. Let $s_i = (P'_i, \widetilde{P}_i, R_i)$ for $i \in \{1,2\}$. By the definition of $\mathbb{E}$, it holds that $\widetilde{P}_1 = \widetilde{P}_2$ and therefore that $P_1 = P_2 = \widetilde{P}_1[\mathsf{undef}]$. The desired result follows from letting $P_3 = P_1 = P_2$.

5.4 Evaluation steps preserve meaning

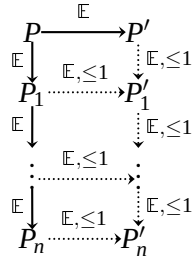
Observe that axiom 4.3(1) follows directly from the definition of result.

We will now prove axiom 4.3(2), i.e., that $\mathbb{E} \subseteq \mathrm{MP}$, directly from axiom 4.2 and the definition of result. Suppose that $P \xrightarrow{s,\mathbb{E}} P'$. $\mathsf{result}(P)$ can be diverges, halt or undef, as can be $\mathsf{result}(P')$.

- Let's first prove that $\mathsf{result}(P) = \mathsf{diverges} \iff \mathsf{result}(P') = \mathsf{diverges}$

  - if $\mathsf{result}(P) = \mathsf{diverges}$, then $\neg$has-nf$(\mathbb{E}, P)$. As a consequence, $\neg$has-nf$(\mathbb{E}, P')$ and $\mathsf{result}(P') = \mathsf{diverges}$.
  - Now suppose that $\mathsf{result}(P) \neq \mathsf{diverges}$ then there exists $P_1, ..., P_n$ such that $P \xrightarrow{\mathbb{E}} P_1 \xrightarrow{\mathbb{E}} \ldots \xrightarrow{\mathbb{E}} P_n$ and is-nf$(\mathbb{E}, P_n)$

With the SubComm property, we obtain the following diagram :

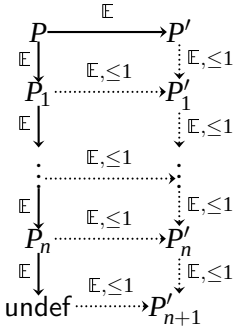


If there exists $j < n$ such that $P_j = P'_j$ then has-nf$(\mathbb{E}, P'_j)$ and consequently has-nf$(\mathbb{E}, P')$ and result$(P') \neq$ diverges.

Otherwise, as is-nf$(\mathbb{E}, P_n)$ there cannot exist $P'_n \neq P_n$ such that $P_n \xrightarrow{\mathbb{E}} P'_n$, and so $P'_n = P_n$, has-nf$(\mathbb{E}, P')$ and result$(P') \neq$ diverges.

Thus result$(P) =$ diverges $\iff$ result$(P') =$ diverges

- Let's now prove that result$(P) =$ undef $\iff$ result$(P') =$ undef
  - If result$(P) \neq$ undef and result$(P') =$ undef then $P \xrightarrow{\mathbb{E}} \!\!\!\!\rightarrow$ undef which is absurd. Thus result$(P') \neq$ undef
  - Now If result$(P) =$ undef then there exists $P_1, ..., P_n$ such that $P \xrightarrow{\mathbb{E}} P_1 \xrightarrow{\mathbb{E}} \ldots \xrightarrow{\mathbb{E}} P_n \xrightarrow{\mathbb{E}}$ undef and we obtain the following diagram with the SubComm property :



If there exists $j \leq n$ such that $P_j = P'_j$ then $P' \xrightarrow{\mathbb{E}} \!\!\!\!\rightarrow$ undef and result$(P') =$ undef.

Otherwise, as is-nf$(\mathbb{E}, \text{undef})$ there cannot exist $P'_{n+1} \neq$ undef such that undef $\xrightarrow{\mathbb{E}} P'_{n+1}$, and so $P'_{n+1} =$ undef and result$(P') =$ undef.

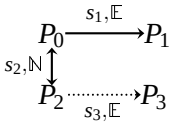
Thus result$(P) =$ undef $\iff$ result$(P') =$ undef

- As result$(P) \in \{$diverges, undef, halt$\}$ and result$(P') \in \{$diverges, undef, halt$\}$, result$(P) =$ halt $\iff$ result$(P') =$ halt.

Thus result$(P) =$ result(P') and $\mathbb{E} \subseteq$ MP.

5.5 Non-evaluation steps preserve evaluation steps

We now need to prove axiom 4.3(3), i.e., the following property:



However, as we will finally want to prove that our entire system is MP, we will also have to look closely at the steps that appear between P_1 and P_3 in order to prove the WB\Std(S) property for a number of sets of steps S in a covering of the full set of steps. Thus, we are not proving just axiom 4.3(3) in this section but also laying the ground for the full proof of meaning preservation.

5.5.1 Part of proof done so far

We will match s_2 with the rules of $\mathbb{N}$ and each case below will correspond to one of the diagrams in figure 2. As $\mathbb{E}$ is mostly deterministic, the direction of s_2 will not be really important. In order to be clear and to make this proof as easy to understand as possible, we will suppose that $P_0 \xrightarrow{s_2} P_2$, and explain in each case how it will similarly work with $P_2 \xrightarrow{s_2} P_0$.

As this is a part of the proof that we have not finished yet, it is incomplete, and our actual work consists in adding the remaining cases to the following enumeration.

- if $s_2 = (P_0, \Box, \mathrm{G}_{1,X_3})$ P_0 is $\mathsf{alloc}\, X_1$; P_o and P_2 is $\mathsf{alloc}\, X_2$; P_o. s_1 can then be :
  - $(P_0, \Box, \mathrm{S}_1)$ if P_o starts with $\mathsf{alloc}\, X'$. In this case we could use $s_3 = (P_2, \Box, \mathrm{S}_1)$ after α-conversion, and we will only have to garbage collect the alloc after (see diagram (1)). It will be exactly the same for the other direction.
  - $(P_0, \Box, \mathrm{D}_{1,\ell})$, $(P_0, \Box, \mathrm{D}_2)$ or $(P_0, \Box, \mathrm{D}_3)$ if P matches their left-hand side. As $X_2 \subseteq \mathsf{FB}(P_o)$, $X_2 \subseteq X_1$ and $x.o \in \mathsf{FB}(P_o)$, whatever the direction we will be able to use $(P_2, \Box, \mathrm{D}_2)$, $(P_2, \Box, \mathrm{D}_3)$ or $(P_2, \Box, \mathrm{D}_4)$ as s_3 and we will be able to do the garbage collection later (see diagram (1)).
  - $(P_o, (\mathsf{alloc}\, X_1; \Box), R)$ and in this case, nothing will change on P_o from P_0 to P_2 or vice versa, so we will be able to use $s_3 = (P_o, (\mathsf{alloc}\, X_2; \Box), R)$ and do the garbage collection later (see diagram (1)). The only potential problem will appear if $R = \mathrm{D}_4$, which could duplicate $\mathsf{alloc}\, X_1$, but it cannot because it only works with BasicValue.
  - it will be the same reasonning with every other $\widetilde{P} \neq \Box$.
  - s_1 cannot possibly be an error rule because of the definition of $\mathbb{S}$ if $P_0 \xrightarrow{s_2} P_2$. In the other direction, it will be the same if $\widetilde{P} \neq \Box$. The only potential problem may appear if $\widetilde{P} = \Box$. The rules can then only be E_6 or E_7 and as $X_2 \subseteq X_1$, this would mean that $P_2 \in \mathsf{ErrorProg}$, which is absurd.

- if $s_2 = (P, (\mathsf{alloc}\, X; \Box), \mathrm{G}_{1,X_3})$ then $P_0 = \mathsf{alloc}\, X; \mathsf{alloc}\, X_1; P_o$ and $P_2 = \mathsf{alloc}\, X; \mathsf{alloc}\, X_2; P_o$. The only possible choice for s_1 is $(P_0, \Box, \mathrm{S}_1)$ and there will also be only one possible choice for $s_3 : (P_2, \Box, \mathrm{S}_1)$ modulo α-conversion. We will then be able to garbage collect later (see diagram (1)). It will be exactly the same for the other direction.

- if $s_2 = (P, (T; \Box), \mathrm{G}_{1,X_3})$ then $P_0 = T; \mathsf{alloc}\, X_1; P_o$ and $P_2 = T; \mathsf{alloc}\, X_2; P_o$. The only possible choice for s_1 is $(P_0, \Box, \mathrm{S}_3)$ and there will also be only one possible choice for $s_3 : (P_2, \Box, \mathrm{S}_3)$ modulo α-conversion. We will then be able to garbage collect later but with the context $\Box$ (see diagram (1)). It will be exactly the same for the other direction.

- if $s_2 = (P, (\mathsf{alloc}\, X; T; \Box), \mathrm{G}_{1,X_3})$, $P_0 = \mathsf{alloc}\, X; T; \mathsf{alloc}\, X_1; P_o$ and $P_2 = \mathsf{alloc}\, X; T; \mathsf{alloc}\, X_2; P_o$. The only possible choice for s_1 is $(P', (\mathsf{alloc}\, X; \Box), \mathrm{S}_3)$ and there will also be only one possible choice for $s_3 : (P', (\mathsf{alloc}\, X; \Box), \mathrm{S}_3)$ modulo α-conversion. We will then be able to garbage collect later but with the context $\mathsf{alloc}\, X; \Box$ (see diagram (1)). It will be exactly the same for the other direction.

- if $s_2 = (P, \widetilde{P}, \mathrm{G}_{1,X_3})$ with $\widetilde{P} \notin \mathcal{E}$, whatever rule we can use on P_0, we will be able to use the same on P_2 (and vice versa) because the beginning will be the same and we will obtain the diagram (1)most of the time. However, if s_1 is a Control rule or a composition of transformers, we may discard or duplicate the portion of code where s_2 was used. In this case we obtain the diagrams (4)or (3).

- if $s_2 = (P_0, \square, \mathrm{G}_{3,T'})$ then $P_0 = \mathsf{alloc}\, X; T; P_o$ and $P_2 = \mathsf{alloc}\, X; T''; P_o$. Whatever the rule we can use on P_0, we will be able to use the same on P_2 because $T'' \subseteq T$ and $\overline{\mathsf{FB}}(\mathsf{Dom}(T')) \cap \mathsf{FB}(P_o) = \varnothing$. We will then be able to do the garbage collection later (see diagram (1)). It will be the same for the other direction.

- if $s_2 = (P, (\mathsf{alloc}\, X_1; \square), \mathrm{G}_{3,T'})$ then $P_0 = \mathsf{alloc}\, X_1; \mathsf{alloc}\, X; T; P_o$, $P_2 = \mathsf{alloc}\, X_1; \mathsf{alloc}\, X; T''; P_o$ and the only posible choice for s_1 is $(P_2, \square, \mathrm{S}_1)$.There will then be only one possible choice for $s_3 : (P_2, \square, \mathrm{S}_1)$ modulo α-conversion. We can then do the garbage collection later and we will obtain the diagram (1). It will be exactly the same for the other direction.

- if $s_2 = (P, (T_1; \square), \mathrm{G}_{3,T'})$ then $P_0 = T_1; \mathsf{alloc}\, X; T; P_o$, $P_2 = T_1; \mathsf{alloc}\, X; T''; P_o$ and the only posible choice for s_1 is $(P_2, \square, \mathrm{S}_3)$.There will then be only one possible choice for $s_3 : (P_2, \square, \mathrm{S}_3)$ modulo α-conversion. We can then do the garbage collection later and we will obtain the diagram (1). It will be exactly the same for the other direction.

- if $s_2 = (P, (\mathsf{alloc}\, X_1; T_1; \square), \mathrm{G}_{3,T'})$ then we will have $P_0 = \mathsf{alloc}\, X_1; T_1; \mathsf{alloc}\, X; T; P_o$ and $P_2 = \mathsf{alloc}\, X_1; T_1; \mathsf{alloc}\, X; T''; P_o$. The only choice for s_1 in this case will be $(P_2, (\mathsf{alloc}\, X_1; \square), \mathrm{S}_3)$ and there will also be only one choice for $s_3 : (P_2, (\mathsf{alloc}\, X_1; \square), \mathrm{S}_3)$ modulo α-conversion. We can then do the garbage collection later and we will obtain the diagram (1). It will be exactly the same for the other direction.

- if $s_2 = (P, \widetilde{P}, \mathrm{G}_{3,T'})$ with $\widetilde{P} \notin \mathcal{E}$, whatever rule we can use on P_0, we will be able to use the same on P_2 (and vice versa) because the beginning will be the same and we will obtain the diagram (1)most of the time. However, if s_1 is a Control rule or a composition of transformers, we may discard or duplicate the portion of code where s_2 was used. In this case we obtain the diagrams (4)or (3).

- if $s_2 = (P, (\mathsf{alloc}\, X; \square), \mathrm{S}_1)$, then $P_0 = \mathsf{alloc}\, X; \mathsf{alloc}\, X_1; \mathsf{alloc}\, X_2; P_o$, $P_2 = \mathsf{alloc}\, X; \mathsf{alloc}\, X_3; P_o$ and the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_1)$. There will then be only one possible choice for $s_3 : (P_2, \square, \mathrm{S}_1)$ modulo α-conversion. We can then finish to merge the alloc 's in one step and we obtain the diagram (5).

- if $s_2 = (P, (T; \square), \mathrm{S}_1)$, then $P_0 = T; \mathsf{alloc}\, X_1; \mathsf{alloc}\, X_2; P_o$, $P_2 = T; \mathsf{alloc}\, X_3; P_o$ and the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_3)$. There will then be only one possible choice for $s_3 : (P_2, \square, \mathrm{S}_3)$ modulo α-conversion. Once we have switched the first alloc with T, we have to switch the second one and then to merge them, which gives diagram (7). It will be the same in the other direcion.

- if $s_2 = (P, (\mathsf{alloc}\, X; T; \square), \mathrm{S}_1)$, then we have $P_0 = \mathsf{alloc}\, X; T; \mathsf{alloc}\, X_1; \mathsf{alloc}\, X_2; P_o$ and $P_2 = \mathsf{alloc}\, X; T; \mathsf{alloc}\, X_3; P_o$. In this case, the only choice for s_1 will be $(P_0, (\mathsf{alloc}\, X; \square), \mathrm{S}_3)$. There will then be only one possible choice for $s_3 : (P_2, (\mathsf{alloc}\, X; \square), \mathrm{S}_3)$ modulo α-conversion. Once we have switched $\mathsf{alloc}\, X_1$ with T, we will have to merge it with $\mathsf{alloc}\, X$, then switch $\mathsf{alloc}\, X_2$ with T and finally merge again. But we will also have to merge the two alloc 's in P_2 to obtain the same program, which gives us the diagram (9)and it will be the same in the other direction.

- if $s_2 = (P_0, \square, \mathrm{S}_2) \in \mathbb{N}$, then $P_0 = T_0; T_1; P_o$, $P_2 = T_{0,1}; P_o$. In this case, the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_{7,\ell})$, but as $\neg\mathsf{ComposeBlocked}(T_0, T_1)$, ℓ will remain the same in $T_{0,1}$ and we will be able to do $s_3 = (P_2, \square, \mathrm{S}_{7,\ell})$. A succession of S_4, S_5, S_6, S_7, S_1, S_2 and S_3 on P_3 and P_1 will then lead to the same program (see diagram (8)).

- if $s_2 = (P, (T_0; \square), \mathrm{S}_2)$, then $P_0 = T_0; T_1; T_2; P_o$ and $P_2 = T_0; T_{1,2}; P_o$.

  - if $\neg(\mathsf{ComposeBlocked}(T_0, T_1)$ or $\mathsf{ComposeUndef}(T_0, T_1))$ then the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_2)$. We then cannot have $\mathsf{ComposeBlocked}(T_0, T_{1,2})$ because we have $\neg\mathsf{ComposeBlocked}(T_1, T_2)$. Either $\mathsf{ComposeUndef}(T_0, T_{1,2})$, which means that the only possible choice for s_3 is $(P_2, \square, \mathrm{E}_{11})$ and that we will have $\mathsf{ComposeUndef}(T_{0,1}, T_2)$ leading to the diagram (2). Or we can use $s_3 = (P_2, \square, \mathrm{S}_2)$ and we will walso be able to merge the remaining transformers in P_1, which will give us the diagram (5).
  - if $\mathsf{ComposeBlocked}(T_0, T_1)$, then the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_{7,\ell})$ and there will only be one possible choice for $s_3 : (P_2, \square, \mathrm{S}_{7,\ell})$. A succession of S_4, S_5, S_6, S_7, S_1, S_2 and S_3 on P_3 and P_1 will then lead to the same program (see diagram (8)).
  - we cannot have $\mathsf{ComposeUndef}(T_0, T_1)$ because it would mean that $P_O \in \mathsf{ErrorProg}$, which is absurd.

- if $s_2 = (P, (\mathsf{alloc}\, X; T; \square), \mathrm{S}_2)$, then $P_0 = \mathsf{alloc}\, X; T_0; T_1; T_2; P_o$ and $P_2 = \mathsf{alloc}\, X; T_0; T_{1,2}; P_o$.

  - if $\neg(\mathsf{ComposeBlocked}(T_0, T_1)$ or $\mathsf{ComposeUndef}(T_0, T_1))$ then the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_2)$. At this point, we cannot have $\mathsf{ComposeBlocked}(T_0, T_{1,2})$ because $\neg\mathsf{ComposeBlocked}(T_1, T_2)$. Either $\mathsf{ComposeUndef}(T_0, T_{1,2})$, which means that the only possible choice for s_3 is $(P_2, (\mathsf{alloc}\, X; \square), \mathrm{E}_{11})$ and that we also will have $\mathsf{ComposeUndef}(T_{0,1}, T_2)$ leading to the diagram (2). Or we will be able to use $s_3 = (P_2, (\mathsf{alloc}\, X; \square), \mathrm{S}_2)$ and we will also be able to merge the remaining transformers in P_1, which will give us the diagram (5).
  - if $\mathsf{ComposeBlocked}(T_0, T_1)$, then the possible choice for s_1 will be $(P_0, (\mathsf{alloc}\, X; \square), \mathrm{S}_{7,\ell})$ and there will only be one possible choice for $s_3 : (P_2, (\mathsf{alloc}\, X; \square), \mathrm{S}_{7,\ell})$. A succession of S_4, S_5, S_6, S_7, S_1, S_2 and S_3 on P_3 and P_1 will then lead to the same program (see diagram (8)).
  - we cannot have $\mathsf{ComposeUndef}(T_0, T_1)$ because it would mean that $P \in \mathsf{ErrorProg}$, which is absurd.

- if $s_2 = (P, (T; \square), \mathrm{S}_3)$, then $P_0 = T_0; T_1; \mathsf{alloc}\, X; P_o$ and $P_2 = T_0; \mathsf{alloc}\, X; T_1; P_o$.

  - if $\neg(\mathsf{ComposeBlocked}(T_0, T_1)$ or $\mathsf{ComposeUndef}(T_0, T_1))$ then the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_2)$. There will then be only one possible choice for $s_3 : (P_2, \square, \mathrm{S}_3)$ modulo α-conversion. We then have to merge the transformers in P_3, which will still be possible because the contents will not have changed, and to switch the alloc and the transformer in P_1, which gives the diagram (6). It will be exactly the same in the other direction.
  - if $\mathsf{ComposeBlocked}(T_0, T_1)$, then the only possible choice for s_1 will be $(P_0, \square, \mathrm{S}_{7,\ell})$, and there will still be only one possible choice for $s_3 : (P_2, \square, \mathrm{S}_3)$. A succession of S_4, S_5, S_6, S_7, S_1, S_2 and S_3 on P_3 and P_1 will then lead to the same program (see diagram (8)).
  - if $\mathsf{ComposeUndef}(T_0, T_1)$, then the only possible choice for s_1 will be $(P_0, \square, \mathrm{E}_{11})$, and there will still be only one possible choice for $s_3 : (P_2, \square, \mathrm{S}_3)$. We can then use $(P_3, (\mathsf{alloc}\, X; \square), \mathrm{E}_{11})$ to join them and we will obtain the diagram (2).

- if $s_2 = (P, (\mathsf{alloc}\, X_1; T; \Box), \mathrm{S}_3)$, then we will have $P_0 = \mathsf{alloc}\, X_1; T_0; T_1; \mathsf{alloc}\, X; P_o$ and $P_2 = \mathsf{alloc}\, X_1; T_0; \mathsf{alloc}\, X; T_1; P_o$.
  - if $\neg(\mathsf{ComposeBlocked}(T_0, T_1)$ or $\mathsf{ComposeUndef}(T_0, T_1))$ then the only possible choice for s_1 will be $(P_0, (\mathsf{alloc}\, X_1; \Box), \mathrm{S}_2)$. There will then be only one possible choice for s_3 : $(P_2, (\mathsf{alloc}\, X_1; \Box), \mathrm{S}_3)$ modulo α-conversion. We then have to merge the alloc 's and transformers in P_3, which will still be possible because the contents will not have changed, to switch the alloc and the transformer and then to merge the alloc 's in P_1, which gives the diagram (10). It will be exactly the same in the other direction.
  - if $\mathsf{ComposeBlocked}(T_0, T_1)$, then the only choice for s_1 will be $(P_0, (\mathsf{alloc}\, X_1; \Box), \mathrm{S}_{7,\ell})$, and there will still be only one possible choice for $s_3 : (P_2, (\mathsf{alloc}\, X_1; \Box), \mathrm{S}_3)$. A succession of S_4, S_5, S_6, S_7, S_1, S_2 and S_3 on P_3 and P_1 will then lead to the same program (see diagram (8)).
  - if $\mathsf{ComposeUndef}(T_0, T_1)$, then the only choice for s_1 will be $(P_0, (\mathsf{alloc}\, X_1; \Box), \mathrm{E}_{11})$, and there will still be only one possible choice for $s_3 : (P_2, \Box, \mathrm{S}_3)$. We can then merge the alloc 's and apply the error rule in P_3 to obtain the same result, which will give the diagram (13).
- if $s_2 = (P, (\mathsf{alloc}\, X_1; T; \Box), \mathrm{S}_3)$, then we will have $P_0 = \mathsf{alloc}\, X_1; T; T_1; \mathsf{alloc}\, X; P_o$ and $P_2 = \mathsf{alloc}\, X_1; T; \mathsf{alloc}\, X; T_1; P_o$. Consequently, the only choice for s_1 will be $(P_0, (\mathsf{alloc}\, X_1; \Box), \mathrm{S}_2)$. There will then be only one choice for $s_3 : (P_2, (\mathsf{alloc}\, X_1; \Box), \mathrm{S}_3)$ modulo α-conversion. We then have to merge the alloc 's and the transformers in P_3 and to switch the $\mathsf{alloc}\, X$ and the transformer, then merge the alloc 's in P_1, which gives the diagram (10). It will be exactly the same in the other direction.
- if $s_2 = (P, \Box, \mathrm{S}_{7,\ell})$ or $(P, (\mathsf{alloc}\, X; \Box), \mathrm{S}_{7,\ell_i})$, then $P_0 = (\mathsf{alloc}\, X); \overbrace{\{\ell_i := (S_1; \ldots; S_n; V)\} \uplus T_1}^{T_2}; P_o$ and $P_2 = (\mathsf{alloc}\, X); S_1; \ldots; S_n; \{\ell_i := V\} \uplus T_1; P_o$ but there exists $\ell_k \in T_2$ with $k < i$ such that $T_2(\ell_k) = W_k$. We can then apply S_1 or S_2 or S_3 or S_4 or S_5 or S_6 or S_7 on P_2 and join P_3 and P_1 by a succession of the same rules, which gives the diagram (8).
- if $s_2 = (P, (T_0; \Box), \mathrm{S}_{7,\ell})$, then we will have $P_0 = T_0; \overbrace{\{\ell := (S_1; \ldots; S_n; V)\} \uplus T_1}^{T_2}; P_o$ and $P_2 = T_0; S_1; \ldots; S_n; \{\ell := V\} \uplus T_1; P_o$.
  - if $\neg(\mathsf{ComposeUndef}(T_0, T_2)$ or $\mathsf{ComposeBlocked}(T_0, T_2))$, then the only possible choice s_1 will be $(P_0, \Box, \mathrm{S}_2)$. As $(S_1; \ldots; S_n; V)$ is a SafeValue, S_1 must be a alloc , and there will be only one choice for s_3 : $(P_2, \Box, \mathrm{S}_3)$ modulo α-conversion. Once again, as $(S_1; \ldots; S_n; V)$ was a SafeValue, we will be able use the rules S_4, S_5, S_4, S_7, S_1, S_2 and S_3 on P_3 and P_1 to finnally obtain $\mathsf{alloc}\, X_f; T_f$ and diagram (8). It will be exactly the same in the other direction.
  - if $\mathsf{ComposeBlocked}(T_0, T_2)$, then the only possible choice for s_1 is $(P_0, \Box, \mathrm{S}_{7,\ell'})$. s_3 will be the same as above, and we will also obtain diagram (8).
  - if $\mathsf{ComposeUndef}(T_0, T_2)$, then the only possible choice for s_1 is $(P_0, \Box, \mathrm{E}_{11})$. As we cannot remove any static location from T_0 or T_1 by only doing the rules S_4, S_5, S_4, S_7, S_1, S_2 and S_3, there will still be a point after P_3 where we can apply the same error rule, and we will then obtain the diagram (14).
- if $s_2 = (P, (\mathsf{alloc}\, X; T_0; \Box), \mathrm{S}_{7,\ell})$, then $P_0 = \mathsf{alloc}\, X; T_0; \overbrace{\{\ell := (S_1; \ldots; S_n; V)\} \uplus T_1}^{T_2}; P_o$ and $P_2 = \mathsf{alloc}\, X; T_0; S_1; \ldots; S_n; \{\ell := V\} \uplus T_1; P_o$.

– if $\neg(\mathsf{ComposeUndef}(T_0, T_2)$ or $\mathsf{ComposeBlocked}(T_0, T_2))$, then the only possible choice s_1 will be $(P_0, (\mathsf{alloc}\ X; \Box), \mathrm{S}_2)$. As $(S_1; \ldots; S_n; V)$ is a SafeValue, S_1 must be a alloc , and there will be only one choice for $s_3 : (P_2, (\mathsf{alloc}\ X; \Box), \mathrm{S}_3)$ modulo α-conversion. Once again, as $(S_1; \ldots; S_n; V)$ was a SafeValue, we will be able use the rules S_4, S_5, S_4, S_7, S_1, S_2 and S_3 on P_3 and P_1 to finnally obtain $\mathsf{alloc}\ X_f; T_f$ and diagram (8). It will be exactly the same in the other direction.

– if $\mathsf{ComposeBlocked}(T_0, T_2)$, then the only possible choice for s_1 is $(P_0, (\mathsf{alloc}\ X; \Box), \mathrm{S}_{7,\ell'})$. s_3 will be the same as above, and we will also obtain diagram (8).

– if $\mathsf{ComposeUndef}(T_0, T_2)$, then the only possible choice for s_1 is $(P_0, \Box, \mathrm{E}_{11})$. As we cannot remove any static location from T_2 or T_1 by only doing the rules S_4, S_5, S_4, S_7, S_1, S_2 and S_3, there will still be a point after P_3 where we can apply the same error rule, and we will then obtain the diagram (14).

5.5.2 Discussion of part of proof remaining to be done

What follows is just an informal skeleton of what remains to be written here in order to have the complete proof.

First, in order not just to prove axiom 4.3(3) but also to apply the WB\Std method, we must define a covering of the set of steps. Then, we must use our analysis of the diagrams to justify for each member S of the covering that $\mathsf{LConf}(\mathcal{S})$, $\mathsf{WLP1}(\mathcal{S})$, and $\mathsf{NE}(\mathcal{S})$. From what we have seen so far, all of these properties look like they will hold, but we have not carefully checked every single case. Finally, completely separately from the analysis of diagrams, we must prove $\mathsf{Trm}(\mathcal{S})$. Our preliminary investigation makes us think the termination arguments will not be too difficult, as long as we keep each member S of the covering as small as possible. It currently looks like there will be no problem in keeping the members of the covering small enough.

Now we discuss which diagrams still need to be analyzed:

- s_2 could use the rule S_4 with an evaluation context if there was a conflict with $S_{7,\ell}$, which cannot be the case.

- s_2 can then use the rule S_4 with a non-evaluation context, which would not change anything to the beginning of the program, and in this case whatever is used before s_2 can also be used after s_2. We will be able to join the remaining programs with the same rule as s_2, which give the diagram (1).

- s_2 can use the rule S_5 only with a non-evaluation context and the reasoning will be exactly the same as above.

- Once again, if s_2 uses S_6, it will be the same as above because it can only happen with non-evaluation contexts. The only difference will be that more steps will be needed to join the remaining programs, and the diagram will be more like the diagram (8).

- s_2 can use the rule $\mathrm{D}_{1,\ell}$ with any context and in this case we can have

  – s_1 using rule $\mathrm{S}_{7,\ell'}$, which gives $s_3 = s_1$ and then a diagram quite similar to diagram (8).

  – s_1 using rule S_2 which gives $s_3 = s_1$ and diagram (1).

  – in any case, as the rule $\mathrm{D}_{1,\ell}$ doesn't change the shape of the program, whatever rule can be applied as s_1, we will be able to do quite the same for s_3.

- s_2 can use the rule D_2 with $\square$ and still be in $\mathbb{N}$ if there exists some s_1, which can only use S_5, and in this case, s_3 can use S_4 and we can join them using $D_{1,\ell}$, which gives a new diagram.

- s_2 can use the rule D_2 with any other context, and as it doesn't change the shape of the program, whatever can be applied as s_1, we will be able to do quite the same for s_3.

- It will be quite the same with D_3, with perhaps one more interaction between D_3 and S_2.

- D_4 can be used with the contexts $\square$ and $(\mathsf{alloc}\, X; \square)$ and still be in $\mathbb{N}$ if there exists a conflict with another rule (S_5 or $S_{7,\ell}$ for example). We will always be able to use quite the same rule after it (S_4 or $S_{7,\ell}$) and then join the edges with D_4 itself and other S-rules.

- D_4 can also be used in $\mathbb{N}$ with any other context, but it will be quite the same.

- s_2 can also use the rules C_1, C_2 and C_3 with non-evaluation contexts, but it won't change the beginning of the program and whatever we can use before s_2, we will be able to do the same after. We will in these cases obtain the diagram (1).

- if s_2 uses the rules A_1 or A_2 after a transformer (contexts $(T; \square)$ or $(\mathsf{alloc}\, X; T; \square)$), s_1 can use the rule S_2, but we will be able to use the same as s_1 after s_2, and join the remaining programs with A_1 or A_2.

- if s_2 uses the rules A_1 or A_2 with a non-evaluation context, whatever can be used as s_1, we will be able to use the same as s_3 and we will have the diagram (1).

- if s_2 uses $T_{\widehat{P}}$:

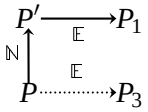
  - there can be a "conflict" with S_2 if they both work on the same transformer, but because of the definition of S_2, there cannot be any problem and we will have diagram (1).
  - there cannot be any conflict with D-rules, because they do not work on safe values.
  - there can be another "conflict" with $S_{7,\ell}$ if they work on the same location ℓ, but because of the definition of $T_{\widehat{P}}$, we will still be able to do $S_{7,\ell}$ after s_2 and to join the edges with multiple uses of S-rules and one use of $T_{\widehat{P}}$.

5.6 Non-evaluation steps on $\mathbb{E}$-normal forms preserve meaning

We now prove axiom 4.3(4), i.e., that if $P \xrightarrow{\mathbb{N}} P'$ and is-nf$(\mathbb{E}, P)$, then $\mathsf{result}(P) = \mathsf{result}(P')$.

Suppose that $P \xrightarrow{\mathbb{N}} P'$ and is-nf$(\mathbb{E}, P)$, and we will try to prove the desired result.

First, we prove that is-nf$(\mathbb{E}, P')$. Suppose $\neg$is-nf$(\mathbb{E}, P')$. Then there exists P_1 such that $P' \xrightarrow{\mathbb{E}} P_1$, but we have already proven that



which leads to the contradiction that $\neg$is-nf$(\mathbb{E}, P)$. Thus, is-nf$(\mathbb{E}, P')$.

By the definition of result, it follows that $\mathsf{result}(P') \in \{\mathsf{undef}, \mathsf{halt}\}$.

Suppose that $\mathsf{result}(P) = \mathsf{undef}$. Because P is an evaluation normal form, it must hold that $P = \mathsf{undef}$. However, there is no step s such that $\mathsf{endpoints}(s) = (\mathsf{undef}, P'')$ for any P'', contradicting that $P \xrightarrow{\mathbb{N}} P'$. Thus, $\mathsf{result}(P) = \mathsf{halt}$.

Now we need to prove that $\mathsf{result}(P) = \mathsf{halt}$, i.e., that $\mathsf{result}(P) \neq \mathsf{undef}$. Unfortunately, the reasoning for this case has not yet been done.

Assuming we reach this point without a proof gap, we therefore conclude the desired result.

6 Future Work

It is quite clear that this work is not finished. As it was said before, it is still going on in order to submit a short version to ESOP '06 (the European Symposium on Programming) on 2005-10-14 and other visits are planned in Edinburgh in order to complete this document. Because this *stage* was quite short, we have done many restrictions on the i-calculus and on the rewrite rules presented here in order to finish something on time for ESOP '06. Consequently, many generalizations are still possible and may be the source of even more work together.

References

[1] J. B. Wells, Detlef Plump, and Fairouz Kamareddine. Diagrams for meaning preservation. pages 88–106. A long version is [2].

[2] J. B. Wells, Detlef Plump, and Fairouz Kamareddine. Diagrams for meaning preservation. Long version of [1], April 2003.

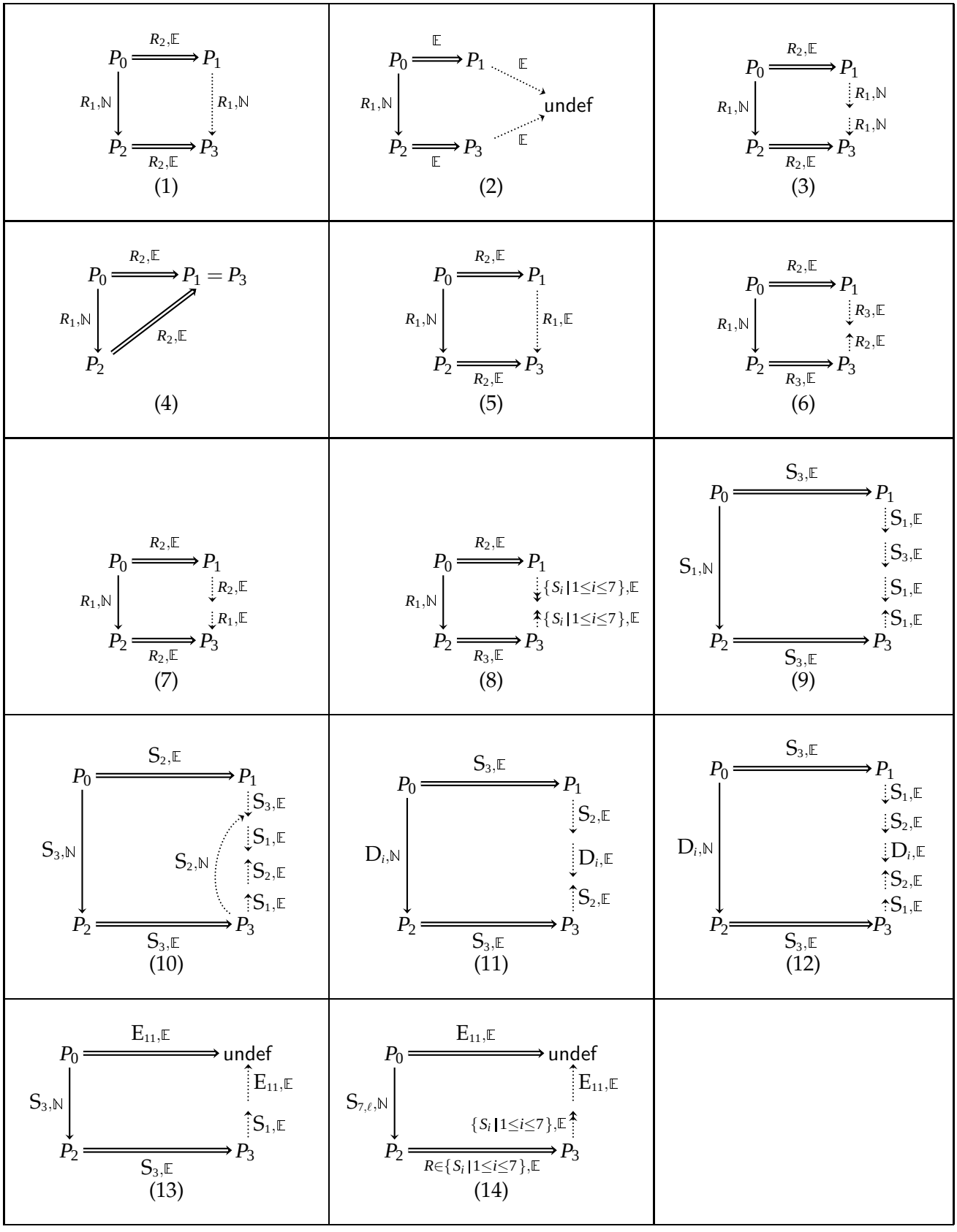


Figure 2: Diagrams