Formal Models of Distributed Memory Management

Cristian Ungureanu and Benjamin Goldberg
Department of Computer Science*
New York University
email: {ungurean,goldberg}@cs.nyu.edu

Abstract

We develop an abstract model of memory management in distributed systems. The model is low-level enough so that we can express communication, allocation and garbage collection, but otherwise hides many of the lower-level details of an actual implementation.

Recently, such formal models have been developed for memory management in a functional, sequential setting [8]. The models are rewriting systems whose terms are programs. Programs have both the "code" (control string) and the "store" syntactically apparent. Evaluation is expressed as conditional rewriting and includes store operations. Garbage collection becomes a rewriting relation that removes part of the store without affecting the behavior of the program.

Distribution adds another dimension to an already complex problem. By using techniques developed for communicating and concurrent systems [7], we extend their work to a distributed environment. Sending and receiving messages is also made apparent at the syntactic level. A very general garbage collection rule based on reachability is introduced and proved correct. Now proving correct a specific collection strategy is reduced to showing that the relation between programs defined by the strategy is a subrelation of the general relation. Any actual implementation which is capable of providing the transitions (including their atomicity constraints) specified by the strategy is therefore correct.

This model allows us to specify and prove correct in a compact manner two garbage collectors; the first one does a simple garbage collection local to a node. The second garbage collector uses migration of data in order to be able to reclaim inter-node cyclic garbage.

1 Introduction

Automatic memory management, or garbage collection, is a valuable service, significantly freeing programmer's resources. Programmers can rely on the language implementation to find and deallocate unneeded objects while also ensuring memory safety: no program will use dangling pointers. Although garbage collectors come with their problems (e.g. run-time costs, possibly additional synchronization costs in concurrent systems) the benefits usually outweigh the drawbacks if the collector indeed ensures memory safety. Unfortunately, the proof that the garbage collector achieves that is very rarely done in

*address: 251 Mercer Street, New York, NY 10012 a satisfactory manner; this happens not only because of the complexity of the strategies used, but also because of the lack of a simple model of memory operations.

In this paper, we are presenting such a model. Starting from the λ_v-$\mathbf{S}$ calculus of Felleisen and Hieb [4], and from Milner's CCS [7], we introduce a language, $\lambda_{||}$, which roughly corresponds to a distributed, impure functional language.

In Section 2, we present the language $\lambda_{||}$ with a rewriting semantics that makes allocation and communication explicit. The semantics defined allows us to use many of the proof techniques developed for CCS. In Section 3, we define the semantic notion of garbage and introduce and prove correct the *free-variable* garbage collection rule which models trace-based collectors. In Section 4 we provide two "implementations" at the syntactic level: one that corresponds to a simple local garbage collector which scans a local heap starting from the local "stack" and the "incoming reference list", and another one which is able to collect garbage cycles which span multiple nodes by migrating objects not referenced locally. We prove that they are subrelations of the garbage collection relation, hence their correctness follows. Section 5 discusses related work and Section 6 presents a summary and future work.

2 The programming language $\lambda_{||}$

Our model of memory management is based on a language and its accompanying semantics. To be adequate for this task the language has to be expressive, with a natural allocation model and a natural concurrency model. In order to have a manageable formalism (fewer cases, shorter proofs, etc.) we have only included essential constructs which permit us to make our point.

While our language doesn't correspond exactly to an existing language, it roughly corresponds to an impure functional language with threads in a distributed environment. The communication mechanism it uses is that of CCS: communication consists of synchronously sending and receiving a value via a port. The allowable expressions, which include assignment, and their evaluation rules are based on λ_v-$\mathbf{S}$.

2.1 Syntax

A *program* is a collection of one or more *processes* running in parallel; they are intended to model the nodes, or sites, of a distributed system. A process P consists of a *thread* T and the local store it has access to, also called *heap* H. The heap contains bindings which can be mutually recursive, since we allow modification of bindings through assignment. The thread part of the process, which may be thought of as the code of the process, may contain sub-threads, all of which share the local memory. The syntax is presented in Figure 1.

Essentially, the thread specifies what communication is possible. Using CCS terminology, the syntactic forms a thread can take are called *combinators*. We have the following combinators: idle thread, prefix, composition, sum, conditional, and recursion. The intention behind these combinators is that the idle thread ε cannot do any actions; prefix allows the specification of a communication (send or receive), and composition allows threads to run in parallel. The sum combinator allows a nondeterministic choice between the two component threads. Recursion allows a thread to be specified as the unique solution of an equation containing a *thread variable*. Not all such equations have unique solutions. If the solution

(*variables*)	$x, y, z \in$	*Var*				
(*integers*)	$i \in$	*Int*	$::= \cdots -1 \mid 0 \mid 1 \mid 2 \mid \cdots$			
(*expressions*)	e	*Exp*	$x \mid i \mid \lambda x.e \mid e_1\ e_2 \mid x := e$			
(*heap values*)	h	*Hval*	$i \mid \lambda x.e$			
(*heaps*)	H	*Heap*	$\{x_1 = h_1, x_2 = h_2 \ldots\}$			
(*ports*)	$\alpha, \beta \in$	*Port*				
(*thread var*)	$X, Y \in$	*TVar*				
(*threads*)	T, U	*Thread*	ε	idle thread	$\mid T_1 + T_2$	sum
			$T_1 \parallel T_2$	composition	$\mid X$	thread variable
			$\alpha?x.T$	prefix (receive)	$\mid$ **fix** $\{X = T\}$	recursion
			$\alpha!e.T$	prefix (send)	$\mid$ **if** x **then** T **else** U	conditional
(*process*)	P, Q	*Process*	$\langle\langle H, T\rangle\rangle$			
(*programs*)	E, F	*Prog*	$P \mid E \odot F$			

Figure 1: Syntax of $\lambda_{||}$

is unique,[1] it is denoted by **fix** $\{X = T\}$. We will precisely describe the behavior of all the combinators when we present the semantics of the language.

We are only providing enough combinators to achieve the desired expressibility of the language. As in CCS, we can add relabeling and restriction, but that would make our models more complicated than is necessary. In CCS, relabeling is only a convenience: programs can be written more compactly by reusing components. Restriction of ports makes it possible to isolate transitions internal to a component of the program from interference from other parts of the program. Because our language does not allow us to hide transitions, we have the extra burden to take these transitions into account when proving bisimilarity of programs. In the programs written in $\lambda_{||}$, in order to achieve a similar effect, we can give unique names to the ports involved.

In $\lambda_{||}$, the values are either integers i or abstractions $\lambda x.e$. Expressions may be values, applications $(e_1\ e_2)$ or assignments $x := e$. The assignment produces both a value (that of e) and a side effect.

The heap is a set of pairs, also called bindings. The pairs consists of a variable and a value. The variable in such a pair can be thought of as the location where the value is stored, rather than a program variable. In all the rules requiring allocation (*Alloc*, *App*, and α_v) we have to ensure that a fresh variable is chosen for a binding; this in effect guarantees the uniqness of "locations" in memory. On the other hand, in the control string we expect to work with "program variables". However, since each program variable is allocated at a unique location, we can keep the correspondence between a program variable and its location by replacing all occurrences of that variable in the control string with the heap variable (the location). We could have chosen to represent program variables and locations by different syntactic categories, but since no confusion is possible we preferred to have a simpler syntax. This notation is also consistent with that found in the referenced papers. *Dom*(H) denotes the bound variables of H, and *Rng*(H) denotes the values bound in H. A variable may be bound to only a single value. Consequently,

[1] A sufficient condition for an equation $X = T$ to have a unique solution is for the variable X to only occur in T in a prefix combinator.

a heap can also be considered a finite function. Moreover, we require that a well formed program has the domains of all component heaps disjoint (a variable is bound in at most one heap).

Notation: $\Lambda_{||}$, $\Lambda^0_{||}$, $\uplus$.
$\Lambda_{||}$ is the set of all well formed programs. $\Lambda^0_{||}$ is the set of all closed programs (no free variables or free process variables). For a definition of free variables see Figure 5. The union of two heaps H_1 and H_2 with disjoint *domains*, is denoted by $H_1 \uplus H_2$.

2.2 Semantics

Operational semantics is usually represented by a state transition system. In our case, the state of the program is syntactically apparent (it is available in the heap and thread parts of processes). Consequently, our semantics will consist of a number of (transition) relations between programs (see Figure 3). The relations are specified by decomposing the program into an *instruction* and an *evaluation context*, and then giving rewriting rules for each possible instruction, together with the possible side conditions. An "execution step" consists of selecting a pair from the relation such that the program matches its left term; the right term is the resulting program. Informally, we will also say that the program *made* that transition, resulting in the new program. When no rules are applicable, the execution halts.

Because processes need not be contiguous to communicate, we shall use the technique of *labeled transitions* to express the laws of process reduction. Formally, we have a set of labels

$$Lbl = \{\alpha_v | \alpha \in Port, v \in Hval\} \cup \{\overline{\alpha}_v | \alpha \in Port, v \in Hval\} \cup \{comp, comm_l, if_0, if_1, sum_1, sum_2, comm_r\}$$

and each label has associated a transition relation. Note that we have an infinite number of labels α_v and $\overline{\alpha}_v$, one for each possible port–value pair. We will say that, for any α and v, the labels α_v and $\overline{\alpha}_v$ are *complementary*, in notation $\alpha_v \frown \overline{\alpha}_v$.

The definition of these relations is given inductively. The fact that a pair belongs to a relation will be asserted by some rules in case the pair occurs as the conclusion of that rules. A rule can also have hypotheses, possibly requiring that some other pair of programs belongs to the same or different transition relation. In notation, rules start with capital letters (as in If_0) while labels and the relations they define are in small case. A rule without hypotheses is called an *axiom*. The other rules are called *inference rules*. This mode of defining the relations will allow us to prove transition invariants by induction on the length of the proof that the transition is possible. This proof method is called *transition induction* [7].

With the exception of rule $Comm_r$ the left term of the pair is a process (thread–heap pair). The transitions that a process can perform are all determined by the thread expression of that process (by the thread combinators).

In rule $\overline{\alpha}_v$, a thread $\alpha!y.T$ is able to send the value of y at port α, after which it behaves like T. The side condition expresses the fact that the value used in the label must coincide with the value y is bound to. The binding may occur anywhere in the program, not necessarily the local heap. In order to simplify the presentation we did not introduce a transition to signal program errors like "unbound variable error". If no transition rule applies, the program is simply stuck.

Expression evaluation contexts, and instruction expression:

$$\begin{array}{llll}
(\mathit{contexts}) & C[] & \in \mathit{Ctxt} & ::= [] \mid C[]\ e \mid x\ C[] \mid x := C[] \\
(\mathit{instructions}) & I & \in \mathit{Instr} & ::= h \mid x\ y \mid x := y
\end{array}$$

Evaluation rules for expressions:

$$\begin{array}{lll}
(\mathit{Alloc}) & (H, C[h], H') \overset{alloc}{\longmapsto} (H \uplus \{x = h\}, C[x]) & x \notin \mathit{Dom}(H \uplus H') \\
(\mathit{App}) & (H, C[x\ y], H') \overset{app}{\longmapsto} (H \uplus \{z' = h\}, C[e\{z'/z\}]) & (H \uplus H')(x) = \lambda z.e, \\
 & & (H \uplus H')(y) = h, z' \notin \mathit{Dom}(H \uplus H') \\
(\mathit{Assign}) & (H \uplus \{x = h_1\}, C[x := y], H') \overset{assign}{\longmapsto} (H \uplus \{x = h_2\}, C[x]) & (H \uplus H')(y) = h_2
\end{array}$$

Figure 2: Expression evaluation contexts, instructions and evaluation rules for expressions

Program contexts: $\mathcal{E}[] ::= [] \mid \mathcal{E}[] \odot E \mid E \odot \mathcal{E}[]$

Rewriting rules for programs:

$$\begin{array}{lrll}
(\overline{\alpha}_v) & \mathcal{E}[\langle\!\langle H, \alpha!y.T \rangle\!\rangle] & \overset{\overline{\alpha}_v}{\longmapsto} \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle] & (H \uplus \mathit{Heap}(\mathcal{E}[]))(y) = v \\
(\alpha_v) & \mathcal{E}[\langle\!\langle H, \alpha?x.T \rangle\!\rangle] & \overset{\alpha_h}{\longmapsto} \mathcal{E}[\langle\!\langle H \uplus \{x' = h\}, T\{x'/x\} \rangle\!\rangle] & x' \notin \mathit{Dom}(H \uplus \mathit{Heap}(\mathcal{E}[])) \\
(\mathit{If}_1) & \mathcal{E}[\langle\!\langle H, \mathbf{if}\ x\ \mathbf{then}\ T\ \mathbf{else}\ U \rangle\!\rangle] & \overset{if_1}{\longmapsto} \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle] & (H \uplus \mathit{Heap}(\mathcal{E}[]))(x) \neq 0 \\
(\mathit{If}_0) & \mathcal{E}[\langle\!\langle H, \mathbf{if}\ x\ \mathbf{then}\ T\ \mathbf{else}\ U \rangle\!\rangle] & \overset{if_0}{\longmapsto} \mathcal{E}[\langle\!\langle H, U \rangle\!\rangle] & (H \uplus \mathit{Heap}(\mathcal{E}[]))(x) = 0 \\
(\mathit{Sum}) & \mathcal{E}[\langle\!\langle H, T_1 + T_2 \rangle\!\rangle] & \overset{sum_i}{\longmapsto} \mathcal{E}[\langle\!\langle H, T_i \rangle\!\rangle] & i = 1, 2 \\
(\mathit{Comm}_l) & \mathcal{E}[\langle\!\langle H, T \parallel U \rangle\!\rangle] & \overset{comm_l}{\longmapsto} \mathcal{E}[\langle\!\langle H', T' \parallel U' \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle] \overset{a}{\longmapsto} \mathcal{E}[\langle\!\langle H, T' \rangle\!\rangle] \\
 & & & \mathcal{E}[\langle\!\langle H, U \rangle\!\rangle] \overset{b}{\longmapsto} \mathcal{E}[\langle\!\langle H', U' \rangle\!\rangle], a \frown b \\
(\mathit{Comp}) & \mathcal{E}[\langle\!\langle H, \alpha!C[e].T \rangle\!\rangle] & \overset{comp}{\longmapsto} \mathcal{E}[\langle\!\langle H', \alpha!C[e'].T \rangle\!\rangle] & (H, C[e], \mathit{Heap}(\mathcal{E}[])) \overset{exp}{\longmapsto} (H', C[e']), \\
 & & & exp \in \{\mathit{alloc}, \mathit{app}, \mathit{assign}\} \\
(\mathit{Comm}_r) & \mathcal{E}[E \odot F] & \overset{comm_r}{\longmapsto} \mathcal{E}[E' \odot F'] & \mathcal{E}[E \odot F] \overset{a}{\longmapsto} \mathcal{E}[E' \odot F] \\
 & & & \mathcal{E}[E \odot F] \overset{b}{\longmapsto} \mathcal{E}[E \odot F'], a \frown b
\end{array}$$

$$\begin{array}{lrll}
(\mathit{Fix}) & \mathcal{E}[\langle\!\langle H, \mathbf{fix}\ \{Y = T\} \rangle\!\rangle] & \overset{a}{\longmapsto} \mathcal{E}[\langle\!\langle H', T' \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, T[\mathbf{fix}\ \{Y = T\}/Y] \rangle\!\rangle] \overset{a}{\longmapsto} \mathcal{E}[\langle\!\langle H', T' \rangle\!\rangle] \\
(\mathit{Com}_1) & \mathcal{E}[\langle\!\langle H, T \parallel U \rangle\!\rangle] & \overset{a}{\longmapsto} \mathcal{E}[\langle\!\langle H', T' \parallel U \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle] \overset{a}{\longmapsto} \mathcal{E}[\langle\!\langle H', T' \rangle\!\rangle] \\
(\mathit{Com}_2) & \mathcal{E}[\langle\!\langle H, T \parallel U \rangle\!\rangle] & \overset{a}{\longmapsto} \mathcal{E}[\langle\!\langle H', T \parallel U' \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, U \rangle\!\rangle] \overset{a}{\longmapsto} \mathcal{E}[\langle\!\langle H', U' \rangle\!\rangle]
\end{array}$$

where $a \in \{\alpha_v, \overline{\alpha}_v, if_0, if_1, comm_l, comp, sum_1, sum_2\}$.

Figure 3: Process evaluation rules

Programs:		Contexts:	
$\mathit{Heap}(\langle\!\langle H, T \rangle\!\rangle)$	$= H$	$\mathit{Heap}([])$	$= \emptyset$
$\mathit{Heap}(E \odot F)$	$= \mathit{Heap}(E) \uplus \mathit{Heap}(F)$	$\mathit{Heap}(E \odot \mathcal{E}[])$	$= \mathit{Heap}(\mathcal{E}[] \odot E) = \mathit{Heap}(E) \uplus \mathit{Heap}(\mathcal{E}[])$

Figure 4: Definition of function *Heap*

Free variables:

Expressions

$FV(x) = \{x\}$
$FV(i) = \emptyset$
$FV(\lambda x.e) = FV(e) \backslash \{x\}$
$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

Sets of values

$FV(\emptyset) = \emptyset$
$FV(S \uplus \{v\}) = FV(S) \cup FV(v)$

Heaps

$FV(H) = FV(Rng(H)) \backslash Dom(H)$

Threads

$FV(\varepsilon) = \emptyset$
$FV(\alpha?x.T) = FV(T) \backslash \{x\}$
$FV(\alpha!e.T) = FV(e) \cup FV(T)$
$FV(X) = \emptyset$
$FV(\mathbf{fix}\ \{X = T\}) = FV(T)$
$FV(T \parallel U) = FV(T + U) = FV(T) \cup FV(U)$
$FV(\mathbf{if}\ x\ \mathbf{then}\ T\ \mathbf{else}\ U) = \{x\} \cup FV(T) \cup FV(U)$

Processes and Programs

$$\begin{aligned} FV(\langle\!\langle H, T \rangle\!\rangle) &= FV(H) \cup (FV(T) \backslash Dom(H)) \\ FV(E \odot F) &= (FV(E) \cup FV(F)) \backslash \\ &\quad (Dom(Heap(E)) \cup Dom(Heap(F))) \end{aligned}$$

Free process variables:

$FP(\varepsilon) = \emptyset$
$FP(\alpha?x.T) = FP(\alpha!e.T) = FP(T)$
$FP(T \parallel U) = FP(T + U) = FP(T) \cup FP(U)$
$FP(X) = \{X\}$
$FP(\mathbf{fix}\ \{X = T\}) = FP(T) \backslash \{X\}$
$FP(\mathbf{if}\ x\ \mathbf{then}\ T\ \mathbf{else}\ U) = FP(T) \cup FP(U)$

Figure 5: Definition of FV and FP functions

In rule α_v, a thread $\alpha?x.T$ is able to receive at port α a value v. A new variable (x' in this case) is chosen and is bound in the local heap to the value received. The thread continues to behave like T (where the variable x' has been substituted for x). The capture-avoiding substitution of a variable for another is defined for all the syntactic components of programs (see Definition 10). The substitution of x with a new variable is necessary to maintain unique bindings for variables in the presence of recursive threads. Rules If_0 and If_1 say that a thread **if** x **then** T **else** U behaves like U in case that x is bound in some heap to the integer 0, and like T if x is bound to some other value. Rules Sum_1 and Sum_2 say that the thread $T_1 + T_2$ can behave (non-deterministically) as either T_1 or T_2. Rule $Comm_l$, which describes communication local to a node, applies in case that the thread expression is a composition of two threads capable of transitions with complementary labels (sending and receiving a value v at some port α). Note that the local heap is modified.

Transition relation *Comp* describes a computation step, whereby an expression is reduced to a value (in one or more steps). This expression reduction is intended to define a left-to-right call-by-value evaluation order. The expression is decomposed into an evaluation context and an instruction. The decomposition is guaranteed to be unique [8], and is obtained by scanning the expression from left to right and taking as the instruction the first redex encountered. The context is the expression with a hole replacing the redex. Because of this mode of choosing the instruction, a context will always have at the left of the hole only variables, hence the definition of expression contexts $C[]$ in Figure 2. In all the evaluation rules for expression, we need to supply both the local heap H (which may be modified by the transition) and the union H' of all nonlocal heaps (in case that some variable is bound non-locally). The transition rule *alloc* applies when the instruction is a value, and specifies that a binding of a new variable to that value is added to the *local* heap. The value is replaced by the variable in the expression part. Recall that we required that all heaps have disjoint domains. This translates to the side condition we have for this rule: $x \notin Dom(H \uplus H')$. Rule *App* specifies that the parameter of the abstraction must be bound to the value of the argument y. Alpha-conversion is necessary to ensure that no conflicts occur in the heap. In rule (*Assign*), the value of y replaces the value x is bound to.

There are also "compatible closure" rules: *Fix*, Com_1 and Com_2. The two *Com* rules say that if the thread is a composition, and one of the component threads is able to make a transition, then the original thread is able to make a transition with the same label. The *Fix* rule specifies how the unwinding of the recursion is made.

Finally, rule $Comm_r$ specifies that communication can take place between processes E and F if, in the context provided by $\mathcal{E}[]$, they are able to make transitions with complementary labels. Note that an effect of applying the $Comm_r$ rule is that we may obtain "remote pointers". However, they are not syntactically distinguished from "local pointers", and the same rules apply to them. Circularities between different local stores can be obtained by embedding pointers in closures.

As we have already said, the program is partitioned into an instruction and an evaluation context. However, unlike in the sequential case, this partitioning is not unique. Non-determinism can occur directly, as a result of the sum combinator, or indirectly as a result of parallel composition. We can have non-determinism because the order of transitions made by different threads is arbitrary and the language has assignment; a typical example is a that of a thread reading a "global" variable and another thread assigning to that variable a new value. Another source of non-determinism is overlapping redexes: two threads attempt to send different values via some port α. These difficulties have to be dealt with when defining the semantics of the language.

The "evaluation" of programs is just a union of some of the relations described above: α_i and $\overline{\alpha}_i$ which model how the program interacts with the environment, and *comp*, sum_1, sum_2, if_0, if_1, $comm_l$, $comm_r$ which describe how the computation can proceed without such interaction. We will use the following notation:

$$\begin{aligned}\mathbf{R} &= \{\alpha_i | \alpha \in Port, i \in Int\} \cup \{\overline{\alpha}_i | \alpha \in Port, i \in Int\} \cup \{comp, comm_l, if_0, if_1, sum_1, sum_2, comm_r\} \\ \mathbf{R}^+ &= \mathbf{R} \cup \{\alpha_v | \alpha \in Port, v \in Hval\} \cup \{\overline{\alpha}_v | \alpha \in Port, v \in Hval\}\end{aligned}$$

$\mathbf{R}$ is the evaluation relation. We will make use of $\mathbf{R}^+$ in various proofs. Here, α_i stands for the set of port–value pairs where the value can only be an integer. An explanation of why we don't use the full relations α_v and $\overline{\alpha}_v$ will be given after we define observational simulation.

2.3 Observational equivalence

The semantics defined is intended to model the behavior of a program which executes in some *environment* with which it can communicate through messages. But the program can also make transitions, like If_0 which do not involve communication. Two programs which cannot be distinguished by an observer are observational equivalent. The question is: what is considered observable ? We propose that observable is only the communication the program has with the environment. The other transitions are unobservable, also called *internal.*

It remains to decide when two programs communicate with the environment in an equivalent way. Naturally, if both programs send (respectively receive) equivalent values via equivalent ports, they communicate in an equivalent way. In our case the values are integers, and their equivalence is mathematical equality; on ports, the equivalence is the identity relation. Equality is certainly decidable on integers. We disallow communicating λ-abstractions *with the environment*, since equality on functions is not decidable. Note, however, that *internal* communication of closures is possible.

Formally, we define observational simulation with respect to a relation $(\Rightarrow)$ which abstracts from unobservable computation.

Definition 1 *The transition relation $\Rightarrow$ on $\Lambda_{||}$ with labels from $\{\alpha_i, \overline{\alpha}_i\}$ is defined inductively:*

$$\begin{array}{lll}
E \overset{a}{\Rightarrow} E' & \textit{if } E \overset{a}{\longmapsto} E' & \textit{where } a \in \{\alpha_i, \overline{\alpha}_i\} \\
E \overset{a}{\Rightarrow} E' & \textit{if } E \overset{b}{\longmapsto} E'' \textit{ and } E'' \overset{a}{\Rightarrow} E' & \textit{and } b \in \{comp, comm_l, comm_r, if_0, if_1, sum_1, sum_2\}. \\
E \overset{a}{\Rightarrow} E' & \textit{if } E \overset{a}{\Rightarrow} E'' \textit{ and } E'' \overset{b}{\longmapsto} E' &
\end{array}$$

Definition 2 *A relation $\mathcal{R} \subseteq \Lambda_{||} \times \Lambda_{||}$ is an* **observational simulation** *if it satisfies:*

$$E \;\mathcal{R}\; F \textit{ and } E \overset{a}{\Rightarrow} E' \Rightarrow \exists F' s.t.\ F \overset{a}{\Rightarrow} F' \textit{ and } E' \;\mathcal{R}\; F'$$

where $a \in \{\alpha_i, \overline{\alpha}_i\}$. An observational bisimulation *is a symmetric observational simulation.*

Lemma 1 *Let $\mathcal{S}_1, \mathcal{S}_2$ be observational bisimulations. Then the identity relation Id on programs, the inverse relation $\mathcal{S}_1^{-1}$, the composition relation $\mathcal{S}_1\mathcal{S}_2$, and the union relation $\mathcal{S}_1 \cup \mathcal{S}_2$ are all observational bisimulations.*

Definition 3 (Program Equivalence, $\approx$) *Our semantic equivalence on programs is the* coarsest *observational bisimulation:*

$$\approx \;=\; \bigcup\{\mathcal{S} : \mathcal{S} \textit{ is an observational bisimulation }\}$$

Lemma 2 *Relation $\approx$ is an observational bisimulation and also an equivalence on programs.*
Proof *Relation $\approx$ is a union of observational bisimulations; by Lemma 1, $\approx$ is itself an observational bisimulation. Reflexivity, symmetry and transitivity follow from the same lemma.*

3 Garbage collection

Definition 4 (Garbage collection relation) *A relation $\mathcal{R} \subseteq \Lambda_{||} \times \Lambda_{||}$ is called a garbage collection relation if it preserves bisimilarity, and has the potential of collecting garbage:*

$$E \;\mathcal{R}\; F \textit{ iff } E \approx F \textit{ and } Dom(Heap(F)) \subseteq Dom(Heap(E))$$

The garbage collected consists of all the bindings of variables in $Dom(Heap(E)) \setminus Dom(Heap(F))$. F is called a collection *of E.*

Note that the relation leaves open the possibility of a garbage collection algorithm to replace the value a variable is bound to with something else (the integer zero, for example), as long as the programs remain bisimilar. (This corresponds to replacing a pointer with NULL.) This is done in [8] to reclaim space occupied by objects which, although reachable, would never be accessed by the program.

3.1 Free variable rule

The garbage collection algorithms we are going to define are all based on tracing: all the reachable bindings are preserved. Following [4], reachability is modeled by considering the free variables. If a heap binding is reachable, and the value bound contains free variables, the bindings of these variables are reachable.

Since our garbage collectors will leave the thread part of the program intact, we first define a relation to assert exactly this.

Definition 5 (Thread equivalence, $\overset{p}{\approx}$) *is a relation on programs (respectively on program contexts) defined by:*

$$\begin{array}{llll} \langle\!\langle H,T\rangle\!\rangle \overset{p}{\approx} \langle\!\langle H',T\rangle\!\rangle & \forall H,H' \in Heap & [\,] \overset{p}{\approx} [\,] \\ E \odot F \overset{p}{\approx} E' \odot F' & \textit{if } E \overset{p}{\approx} E' \textit{ and } F \overset{p}{\approx} F' & \mathcal{E}[\,] \odot E \overset{p}{\approx} \mathcal{F}[\,] \odot F \textit{ if } \mathcal{E}[\,] \overset{p}{\approx} \mathcal{F}[\,] \textit{ and } E \overset{p}{\approx} F \end{array}$$

Definition 6 (The sub-heap relation, $\subseteq_H$) *$E \subseteq_H F$ iff $Heap(E) \subseteq Heap(F)$.*

Definition 7 (Free variable relation) *The free variable relation $(\overset{fv}{\longmapsto}) \subseteq \Lambda^0_{||} \times \Lambda^0_{||}$ is defined by:*

$$E \overset{fv}{\longmapsto} F \textit{ if } E \overset{p}{\approx} F \textit{ and } F \subseteq_H E$$

Note that this is a relation between *closed* programs. Our goal is to prove that the free variable relation is also a garbage collection relation. In order to show this we need a number of technical lemmas. We plan to proceed as follows: we define *strong bisimulation* on $\Lambda_{||}$, and show that if two programs are strongly bisimilar, then they are observationally bisimilar. We will then show that α-conversion is a strong bisimilarity. We need this result because two identical programs, making an *alloc* transition for example, may choose different fresh variables, leading to syntactically different programs. However, since the resulting programs can be α-converted into each other, the lemma will tell us that they are bisimilar.

We then define strong bisimulation up to $\simeq$, and show that if $\mathcal{S}$ is a strong bisimulation up to $\simeq$, then $\mathcal{S} \subseteq \simeq$; it follows that $\mathcal{S} \subseteq \approx$. Then, it is sufficient to prove that $(\overset{fv}{\longmapsto})$ is a strong bisimulation up to $\simeq$.

Definition 8 (Strong simulation) *A relation $\mathcal{S} \subseteq \Lambda_{||} \times \Lambda_{||}$ is a* strong simulation *if it satisfies:*

$$\forall a \in \mathbf{R},\ E\ \mathcal{S}\ F \textit{ and } E \overset{a}{\longmapsto} E' \Rightarrow \exists F' \textit{ s.t. } F \overset{a}{\longmapsto} F' \textit{ and } E'\ \mathcal{S}\ F'$$

A strong bisimulation *is a symmetric strong simulation.*

Lemma 3 *Let $\mathcal{S}_1, \mathcal{S}_2$ be strong bisimulations. Then: the identity relation Id on programs, the inverse relation $\mathcal{S}_1^{-1}$, the composition relation $\mathcal{S}_1\mathcal{S}_2$, and the union relation $\mathcal{S}_1 \cup \mathcal{S}_2$ are all strong bisimulations.*

Definition 9 *The* **strong equivalence** *relation, $\simeq$, is the* coarsest *strong bisimulation:*

$$\simeq = \bigcup\{\mathcal{S} : \mathcal{S} \text{ is a strong bisimulation }\}$$

Lemma 4 *Relation $\simeq$ is a strong bisimulation and also an equivalence on programs. The result follows from the previous lemma.*

Lemma 5 *Relation $\simeq$ is an observational bisimulation.*
Proof. *We have to show that: $E \simeq F$ and $E \overset{a}{\Rightarrow} E' \Rightarrow \;\exists F'$ s.t. $F \overset{a}{\Rightarrow} F'$ and $E' \simeq F'$, where $a \in \{\alpha_i, \overline{\alpha_i}\}$. We show this by induction on the length of proof of $E \overset{a}{\Rightarrow} E'$.*

- *$E \overset{a}{\Rightarrow} E'$ because $E \overset{a}{\longmapsto} E'$.*
  But $E \simeq F$ implies $\exists F'.F \overset{a}{\longmapsto} F'$ and $E' \simeq F'$. It follows that $F \overset{a}{\Rightarrow} F'$ and $E' \simeq F'$.

- *$E \overset{a}{\Rightarrow} E'$ because $\exists E''.E \overset{b}{\longmapsto} E''$ and $E'' \overset{a}{\Rightarrow} E'$, where $b \in \{comp,\ comm_l,\ comm_r,\ if_0,\ if_1, sum_1, sum_2\}$.*
  But $E \simeq F$ implies $\exists F''.F \overset{b}{\longmapsto} F''$ and $E'' \simeq F''$. Using the induction hypothesis, from $E'' \overset{a}{\Rightarrow} E'$ and $E'' \simeq F''$ follows that $\exists F'.F'' \overset{a}{\Rightarrow} F'$ and $E' \simeq F'$.

- *$E \overset{a}{\Rightarrow} E'$ because $\exists E''.E \overset{a}{\Rightarrow} E''$ and $E'' \overset{b}{\longmapsto} E'$.*
  From the induction hypothesis $\exists F''.F \overset{a}{\Rightarrow} F''$ and $E'' \simeq F''$. From $E'' \simeq F''$ and $E'' \overset{b}{\longmapsto} E'$ follows that $\exists F'.F \overset{b}{\longmapsto} F'$ and $E' \simeq F'$.

Definition 10 (Substitution) *A substitution θ is an* injective *finite map from variables to variables. θ is naturally extended to a function for all syntactic categories in the language in Figure 6.*

Theorem 1 *Let $\mathcal{S} = \{(E, F) | E, F \in \Lambda^0_{||}$ and $\exists\theta.F = \theta E\}$. $\mathcal{S}$ is a strong bisimulation.*
Proof. *We have to show: $\forall a \in \mathbf{R}, (E, F) \in \mathcal{S}$ and $E \overset{a}{\longmapsto} E' \Rightarrow \exists F'$ s.t. $F \overset{a}{\longmapsto} F'$ and $(E', F') \in \mathcal{S}$. First, we will strengthen the induction hypothesis for the particular case of the substitution relation. Remember that $\mathbf{R}^+ = \mathbf{R} \cup \{\alpha_v, \overline{\alpha}_v\}$. We will show that:*

$$\forall a \in \mathbf{R}^+, (E, F) \in \mathcal{S} \text{ and } E \overset{a}{\longmapsto} E' \Rightarrow \exists\theta' \text{ s.t. } F' = \theta' E' \text{ and } F \overset{\theta' a}{\longmapsto} F'.$$

Note that this implies the property we want to prove, since $\forall a \in \mathbf{R}, a = \theta' a$ (the actions in $\mathbf{R}$ don't contain variables). We shall prove this by transition induction.

Case 1: $E \overset{a}{\longmapsto} E'$ is an axiom. In this case, a must be one of $\alpha_v, \overline{\alpha}_v, if_0, if_1, sum_1, sum_2, comp$. This can only happen if $\exists\mathcal{E}[], H, T, H', T'$ s.t. $E \equiv \mathcal{E}[\langle\langle H, T\rangle\rangle]$, and $E' \equiv \mathcal{E}[\langle\langle H', T'\rangle\rangle]$.

- *The derivation is an instance of $\overline{\alpha}_v$*
  $\mathcal{E}[\langle\langle H, \alpha!x.T\rangle\rangle] \overset{\overline{\alpha}_v}{\longmapsto} \mathcal{E}[\langle\langle H, T\rangle\rangle] \equiv E'$, where $v = Heap(E)(x)$. Let θ be such that $F = \theta E$.
  But $F \equiv \theta(\mathcal{E}[\langle\langle H, \alpha!x.T\rangle\rangle]) = \theta(\mathcal{E})[\langle\langle \theta H, \alpha!\theta x.\theta T\rangle\rangle] \overset{\overline{\alpha}_{\theta v}}{\longmapsto} \theta(\mathcal{E})[\langle\langle\theta H, \theta T\rangle\rangle]$. By taking $\theta' = \theta$ it is obvious that $F \overset{\overline{\alpha}_{\theta' v}}{\longmapsto} F'$.

Values:		Heaps:	
$\theta\ i$	$= i$	$\theta\ \emptyset$	$= \emptyset$
$\theta\ \lambda x.e$	$= \lambda\theta\ x.(\theta\ e)$	$\theta\ (H \uplus \{x = v\})$	$= (\theta\ H) \uplus \{\theta\ x = \theta\ v\}$
Labels:		Threads:	
$\theta\ \alpha_v$	$= \alpha_{(\theta\ v)}$	$\theta\ \alpha?x.U$	$= \alpha?\theta\ x.(\theta\ U)$
$\theta\ \overline{\alpha}_v$	$= \overline{\alpha}_{(\theta\ v)}$	$\theta\ \alpha!e.U$	$= \alpha!\theta\ e.(\theta\ U)$
$\theta\ a$	$= a$, otherwise	$\theta\ \varepsilon$	$= \varepsilon$
		$\theta\ X$	$= X$
		$\theta\ U \parallel V$	$= (\theta\ U) \parallel (\theta\ V)$
Processes and programs:		θ **if** x **then** U **else** V	$=$ **if** $\theta\ x$ **then** $\theta\ U$ **else** $\theta\ V$
$\theta\ \langle\!\langle H,T \rangle\!\rangle$	$= \langle\!\langle \theta\ H, \theta\ T \rangle\!\rangle$	$\theta\ U + V$	$= (\theta\ U) + (\theta\ V)$
$\theta\ E \odot F$	$= (\theta\ E) \odot (\theta\ F)$	θ **fix** $\{X = U\}$	$=$ **fix** $\{X = \theta\ U\}$

Figure 6: Definition of substitution

- *If_0, If_1, Sum_1, Sum_2.*
  Since these transitions do not introduce new variables, and their labels are unaffected by substitution, we can take $\theta' = \theta$.

- *α_v*
  $\mathcal{E}[\langle\!\langle H, \alpha?x.T\rangle\!\rangle] \xrightarrow{\alpha_v} \mathcal{E}[\langle\!\langle H \uplus \{x_1 = v\}, T\{x_1/x\}\rangle\!\rangle] \equiv E'$, where $x_1 \notin Dom(Heap(E))$ and let $u = \theta v$.
  $F \equiv \theta(\mathcal{E}[\langle\!\langle H, \alpha?x.T\rangle\!\rangle]) = \theta(\mathcal{E})[\langle\!\langle \theta H, \alpha?\theta x.\theta T\rangle\!\rangle] \xrightarrow{\alpha_u} \theta(\mathcal{E})[\langle\!\langle \theta H \uplus \{x_2 = u\}, (\theta T)\{x_2/x\}\rangle\!\rangle]$,
  where $x_2 \notin Dom(Heap(\theta E))$. We can choose $\theta' = \theta \uplus \{x_1 = x_2\}$. Given that $x_2 \notin Dom(Heap(\theta E))$ and that the programs are closed, it follows that θ' is indeed injective.
  Since $T\{x_1/x\}\{x_2/x_1\} = T\{x_2/x\}$ it follows that $F \xrightarrow{\alpha_u} F' \equiv \theta' E'$, so $(E', F') \in \mathcal{S}$.

- *Comp.*
  - *Alloc. Let x be the fresh variable chosen in the transition $E \xrightarrow{comp} E'$, and x' be the fresh variable chosen in the transition $\theta E \xrightarrow{comp} F'$. If we take $\theta' = \theta \cup \{x = x'\}$ we have $\theta' E' = F'$.*
  - *App is treated similarly to α_v.*
  - *Assign. We can take $\theta' = \theta$.*

Case 2: $E \xrightarrow{a} E'$ because of an inference rule: Fix, Com_1, Com_2, $Comm_l$, $Comm_r$.

- *Com_1. We have $E \equiv \mathcal{E}[\langle\!\langle H, T \parallel U\rangle\!\rangle] \xrightarrow{a} \mathcal{E}[\langle\!\langle H', T' \parallel U\rangle\!\rangle] = E'$ because $\mathcal{E}[\langle\!\langle H, T\rangle\!\rangle] \xrightarrow{a} \mathcal{E}[\langle\!\langle H', T'\rangle\!\rangle]$.*
  Using the induction hypothesis, we have that $\exists \theta''$ s.t. $\theta\mathcal{E}[\langle\!\langle H, T\rangle\!\rangle] \xrightarrow{\theta'' a} \theta''\mathcal{E}[\langle\!\langle H', T'\rangle\!\rangle] \equiv F''$.
  On the other hand, $\theta'' U = \theta U$. If we take $\theta' = \theta''$ the conclusion follows.

- *Com_2, Fix are treated similarly.*

- *$Comm_l$ (and similarly $Comm_r$). We have $E \equiv \mathcal{E}[\langle\!\langle H, T \parallel U\rangle\!\rangle] \xrightarrow{comm_l} \mathcal{E}[\langle\!\langle H', T' \parallel U'\rangle\!\rangle] \equiv E'$ because $\exists \alpha, v$ s.t. (i) $\mathcal{E}[\langle\!\langle H, T\rangle\!\rangle] \xrightarrow{\alpha_v} \mathcal{E}[\langle\!\langle H, T'\rangle\!\rangle]$ and (ii) $\mathcal{E}[\langle\!\langle H, U\rangle\!\rangle] \xrightarrow{\overline{\alpha}_v} \mathcal{E}[\langle\!\langle H', U'\rangle\!\rangle]$.*

Applying the induction hypothesis to $(\mathcal{E}[\langle\langle H,T\rangle\rangle],\theta\mathcal{E}[\langle\langle H,T\rangle\rangle])$ *we have that* $\exists\theta''.\theta\mathcal{E}[\langle\langle H,T\rangle\rangle] \xmapsto{\alpha_v} \theta''\mathcal{E}[\langle\langle H,T'\rangle\rangle]$. *Similarly* $\exists\theta'''.\theta\mathcal{E}[\langle\langle H,U\rangle\rangle] \xmapsto{\overline{\alpha}_v} \theta'''\mathcal{E}[\langle\langle H,U'\rangle\rangle]$. *Moreover,* $\theta = \theta''' \subseteq \theta''$. *It follows that we can choose* $\theta' = \theta''$.

Definition 11 $\mathcal{S}$ *is a strong bisimulation up to* $\simeq$ *if* $E\ \mathcal{S}\ F$ *implies, for all* a*:*
If $E \xmapsto{a} E'$ *then,* $\exists F', F \xmapsto{a} F'$ *and* $E' \simeq \mathcal{S} \simeq F'$.
If $F \xmapsto{a} F'$ *then,* $\exists E', E \xmapsto{a} E'$ *and* $E' \simeq \mathcal{S} \simeq F'$.
Where $\simeq \mathcal{S} \simeq$ *is the composition of the three binary relations* $\simeq$, $\mathcal{S}$ *and* $\simeq$.

Lemma 6 *If* $\mathcal{S}$ *is a strong bisimulation up to* $\simeq$*, then*
(a) $\simeq \mathcal{S} \simeq$ *is a strong bisimulation,*
(b) $\mathcal{S}\ \subseteq\ \simeq$.

Proof *of (a)*
$E \simeq \mathcal{S} \simeq F \equiv \exists E_1, F_1.E \simeq E_1, F \simeq F_1$ *and* $E_1\ \mathcal{S}\ F_1$. *But then,* $\forall a \in \mathbf{R}, E \xmapsto{a} E' \Rightarrow \exists E_1'.E' \xmapsto{a} E_1'$.
Since $\mathcal{S}$ *is a strong bisimulation up to* $\simeq$*, it follows that* $\exists F_1'.F_1 \xmapsto{a} F_1'$ *and* $E_1' \simeq \mathcal{S} \simeq F_1'$.
Finally, because $F_1 \simeq\ F$ *we have that* $\exists F'.F \xmapsto{a} F'$ *and* $F_1' \simeq F'$. *This proves* $\exists F'.E' \simeq \mathcal{S} \simeq F'$.

Proof *of (b)*
$Id \subseteq\ \simeq$*, hence* $Id\ \mathcal{S}\ Id\ \subseteq\ \simeq \mathcal{S} \simeq$. *It follows that* $\mathcal{S}\ \subseteq\ \simeq \mathcal{S} \simeq\ \subseteq\ \simeq$.

Lemma 7 $(\xmapsto{fv})$ *is a strong bisimulation up to* $\simeq$.
The proof of this lemma (in the appendix) crucially uses the fact that the relation is on closed *programs.*

Theorem 2 (Correctness of GC) $(\xmapsto{fv})$ *is a garbage collection relation. The result follows from Lemmas 7, 6 and 5.*

4 Implementation

The garbage collection rule we have given is just a specification of an algorithm, saying what has to be done: partition the program in some way into the useful bindings and the garbage ones. We are now going to present two algorithms to achieve this partitioning.

4.1 The Free-Variable Tracing Algorithm

We have as model a two space copying garbage collector. The collector traverses the graph of the computation (the *from-heap*) putting all reachable objects into a *to-heap.* The traversal is started from a set of roots, and objects reachable but not yet moved to *to-heap* are kept into a *scan set.*

But what are the roots of the computation ? Since we do not want a system-wide copying but only one local to each node, the roots for each node will be the free variables in the thread expression (the local "stack") *and* all variables bound locally but referenced globally.

In our model this is done as follows: first we define the $(\xmapsto{scan})$ relation (see Figure 7). The from-heap is H_f, the scan set is S and the to-heap is H_t. It can be seen that the remote pointers are not followed. Only variables bound in the local heap are added to the scan set. As usual, the $\mapsto\!\!\!\twoheadrightarrow$ relation is the reflexive and transitive closure of $\longmapsto$.

$$(\xmapsto{scan}) \quad (H_f \uplus \{x = h\}, S \uplus \{x\}, H_t) \xmapsto{scan} (H_f, S \cup (FV(h) \cap Dom(H_f)), H_t \uplus \{x = h\})$$
$$(\xmapsto{fva}) \quad \mathcal{E}[\langle\!\langle H, T\rangle\!\rangle] \xmapsto{fva} \mathcal{E}[\langle\!\langle H', T\rangle\!\rangle] \quad \text{if } (H, Dom(H) \cap (FV(T) \cup FV(\mathcal{E}[])), \emptyset) \overset{scan}{\mapsto\!\!\!\twoheadrightarrow} (H'', \emptyset, H')$$

Figure 7: The algorithm *fva*

The algorithm to collect garbage is specified by the relation $(\xmapsto{fva})$. It specifies that scanning is started with the local heap H as the from-heap, the scan set is composed of the roots of the computation, and the to-heap is empty. After scanning is finished (the scan set is empty), the live data is in H', while H'' contains only garbage. The $(\xmapsto{fva})$ relation still doesn't specify how the root set is computed but only what it is composed of. In an actual implementation, the first part, $FV(T)$, is obtained scanning the stack. The other part, $FV(\mathcal{E}[])$, is maintained during computation. This can be done for example by using a variant of reference counting which uses lists of sites referencing an object (instead of a simple count of them). Each site maintains a list of potential incoming and outgoing references, called the *entry table* and *exit table* respectively. Both tables are conservative estimates. When a pointer is exported, the entry table on the local node and the exit table on the remote node are updated (if necessary). Each local GC cleans the exit table of useless entries. In turn, exit tables are used to clean remote entry tables, yielding successively better estimates. More details about this method can be found in [10]

It is important to note that these steps can be described in our framework (the domain of the transition relations can be augmented with sets for these tables). However, including them would have made the presentation much less clear. Theorem 3 establishes the correctness of the algorithm.

Theorem 3 $(\xmapsto{fva}) \subseteq (\xmapsto{fv})$.

Proof. *Suppose* $E \xmapsto{fva} F$; *we have to show that* $E \overset{p}{\approx} F$, $F \subseteq_H E$ *and* $FV(F) = \emptyset$.
The definition of $\subseteq_H$ *can be found in the appendix (definition 6). We first show that the following invariants are preserved whenever* $(H_f, S, H_t) \xmapsto{scan} (H'_f, S', H'_t)$

1. $H_f \cup H_t = H'_f \cup H'_t$. *Obvious.*

2. $S \cup Dom(H_t) \subseteq S' \cup Dom(H'_t)$. *It can be seen that* $\xmapsto{scan}$ *removes a variable* x *from* S *only when it adds a binding* $x = h$ *to* H_t.

3. $[\ FV(Rng(H_t)) \subseteq Dom(H_t) \cup S \cup (FV(H))\ ] \ \Rightarrow\ [\ FV(Rng(H'_t)) \subseteq Dom(H'_t) \cup S' \cup (FV(H))\ ]$
   Consider an arbitrary element $z \in FV(Rng(H'_t))$.

   - *If* $z \in FV(Rng(H_t))$, *the conclusion follows from the induction hypothesis and invariant 2.*
   - *If* $z \in FV(h) \backslash FV(Rng(H_t))$, *we have the following cases:*

$$
\begin{array}{lll}
(scan) & (H_f \uplus \{x = h\}, S \uplus \{x\}, H_t) \overset{scan}{\longmapsto} (H_f, S \cup (FV(h) \cap Dom(H_f)), H_t \uplus \{x = h\}) & \\
 & & \\
(recv_H) & \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle] \overset{recv_{H_r}}{\longmapsto} \mathcal{E}[\langle\!\langle H \uplus H_r, T \rangle\!\rangle] & \text{if } Dom(H) \cap Dom(H_r) = \emptyset \text{ and } H_r \neq 0 \\
(send_H) & \mathcal{E}[\langle\!\langle H_f, T \rangle\!\rangle] \overset{send_{H_r}}{\longmapsto} \mathcal{E}[\langle\!\langle H_t, T \rangle\!\rangle] & \text{if } (H_f, FV(T) \cap Dom(H_f), \emptyset) \overset{scan}{\longmapsto\!\!\!\!\rightarrow} (H_i, \emptyset, H_t) \\
 & & \text{and } (H_i, FV(\mathcal{E}[]) \cap Dom(H_i), \emptyset) \overset{scan}{\longmapsto\!\!\!\!\rightarrow} (H_g, \emptyset, H_r) \\
(fvc) & \mathcal{E}[E] \overset{fvc}{\longmapsto} \mathcal{E}[E'] & \text{if } \mathcal{E}[E] \overset{send_\emptyset}{\longmapsto} \mathcal{E}[E'] \\
(fvc) & \mathcal{E}[E \odot F] \overset{fvc}{\longmapsto} \mathcal{E}[E' \odot F'] & \text{if } \mathcal{E}[E \odot F] \overset{send_H}{\longmapsto} \mathcal{E}[E' \odot F] \text{ and } \mathcal{E}[E \odot F] \overset{recv_H}{\longmapsto} E \odot F'
\end{array}
$$

Figure 8: The second algorithm

- *$z \notin Dom(H)$, but then $z \in (FV(H))$*
- *$z \in Dom(H) = Dom(H_f) \uplus Dom(H_t)$. If $z \in Dom(H_t)$ we are done. If $z \in Dom(H_f)$ then $z \in (FV(H) \cap Dom(H_f))$, hence $z \in S'$.*

$E \overset{p}{\approx} F$ is obvious since the thread part is unaffected by this transition.

$F \subseteq_H E$ is derived easily from invariant 1.

$FV(F) = \emptyset$. Consider $E \equiv \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle]$ and $F \equiv \mathcal{E}[\langle\!\langle H', T \rangle\!\rangle]$,
where $(H, (FV(T) \cup FV(\mathcal{E}[])) \cap H, \emptyset) \overset{scan}{\longmapsto\!\!\!\!\rightarrow} (H'', \emptyset, H')$.

Initially, $H_t = \emptyset$, so $FV(Rng(H_t)) \subseteq Dom(H_t) \cup S \cup (FV(H))$ is trivially satisfied.
Since invariant 3 proves the correctness of the induction step, it follows that $FV(H') \subseteq FV(H) = \emptyset$.

4.2 The second free-variable tracing algorithm

The idea of the second algorithm, presented in Figure 8, is that we can also use object migration to help collect inter-node garbage cycles. We start from the free variables in the local thread. The heap is partitioned in locally accessible data H_t and locally inaccessible data H_i. The latter is further partitioned into data accessible from the entry tables H_r, and garbage H_g. Note that H_r is only referenced from other nodes. Objects in H_r are potentially part of garbage structures (possibly including cycles) spanning multiple nodes. But then, if we migrate[2] H_r, say to a node which references it, the resulting program is bisimilar with the original program. However, the new program has shortened the length of at least one garbage cycle (if it spanned the node garbage collected). Note that in case $H_r = \emptyset$ the $send_\emptyset$ relation coincides with fva. We do not claim that this algorithm *guarantees* the collection of cyclic garbage spanning multiple nodes; improvements of the algorithm to make it guarantee such a collection are beyond the scope of this paper. Here, we are only interested in showing how a proof of correctness can be given using the model developed.

[2]This migration necessitates sending messages to all remote nodes having remote references to that part of the heap, to inform them of the new destination of their references.

Theorem 4 $(\overset{fvc}{\longmapsto}) \subseteq (\overset{fv}{\longmapsto})$

Proof *We have to show that: if* $E \overset{fvc}{\longmapsto} F$ *for some closed* E*, then* $E \overset{p}{\approx} F$*,* $F \subseteq_H E$ *and* F *closed. There are two cases to consider:*

- $E \overset{fvc}{\longmapsto} F$ *because* $\exists \mathcal{E}[], E'$*, s.t.* $E \equiv \mathcal{E}[E'] \overset{send_\emptyset}{\longmapsto} \mathcal{E}[E'']$*. But* $\overset{send_\emptyset}{\longmapsto}$ *coincides with* $\overset{fva}{\longmapsto} \subseteq \overset{fv}{\longmapsto}$*.*

- $E \overset{fvc}{\longmapsto} F$ *because* $\exists \mathcal{E}[], E', F', H_r$ *s.t.*
  $E \equiv \mathcal{E}[E' \odot F']$ *and* $\mathcal{E}[E' \odot F'] \overset{send_{H_r}}{\longmapsto} \mathcal{E}[E'' \odot F']$ *and* $\mathcal{E}[E' \odot F'] \overset{recv_{H_r}}{\longmapsto} \mathcal{E}[E' \odot F'']$*.*
  Let $E' \equiv \langle\!\langle H_f, T \rangle\!\rangle$ *and* $F' \equiv \langle\!\langle H'_f, U \rangle\!\rangle$*. We have:*
  $(H_f, Dom(H_f) \cap FV(T), \emptyset) \overset{scan}{\longmapsto\!\!\!\!\rightarrow} (H_i, \emptyset, H_t)$ *and* $(H_i, Dom(H_f) \cap FV(\mathcal{E}[]), \emptyset) \overset{scan}{\longmapsto\!\!\!\!\rightarrow} (H_g, \emptyset, H_r)$*.*

  Consider $E' \overset{fva}{\longmapsto} E'''$*.*
  $(H_f, Dom(H_f) \cap (FV(T) \cup FV(\mathcal{E}[])), \emptyset) \overset{scan}{\longmapsto\!\!\!\!\rightarrow} (H'', \emptyset, H')$
  It is easily seen that $H' = H_t \cup H_r$*. Using theorem 3,* $\mathcal{E}[E''' \odot F'] \subseteq_H E$ *and* $FV(\mathcal{E}[E''' \odot F']) = \emptyset$*. On the other hand,* $Heap(E''' \odot F') = Heap(E'' \odot F'')$*. The conclusion follows using simple properties of free variables.*

It may appear surprising that we didn't have any deadlock considerations in our proofs. Note, however, that our proofs rely on the transitions being *atomic*. An even lower-level implementation for the second algorithm would have to address how *fvc* can assure that sending and receiving parts of the heap can be achieved atomically, so as to avoid deadlock, or inconsistencies, caused by two nodes simultaneously initiating $\overset{send_H}{\longmapsto}$ towards each other. Such a lower-level implementation might employ for example a simple token protocol for achieving this: before initiating $\overset{send_H}{\longmapsto}$ a node requests the token (only one token exists in the system). When it receives the token, it sends H_r and update messages to all nodes which reference it. After it receives acknowledgments that the data has been received, it releases the token. Thus, liveness is guaranteed. Note that no such considerations are necessary for the first algorithm, since the $\overset{fva}{\longmapsto}$ rule doesn't involve *two* nodes.

It is only these lower-level implementations that would make a sharper distinction between the local and remote communication relations in their proofs.

5 Related Work

Starting from the λ_v calculus developed by Plotkin in [9], Felleisen and Hieb [4] extend it to a calculus, called λ_v-**S**, suitable for reasoning about state and control in programming languages. This provides the theoretical background of the development of a syntactic theory of memory management in [8]. While in [4] an important effort is made in their model to avoid the garbage generated during computation, in order to obtain a "clean" equational reasoning, in [8] the main topic is exactly that garbage. Using the model developed for a typed language with polymorphism, they were able to prove formally that some *reachable* data is garbage (it will never be accessed). This property was previously reported [5], but its proof was less formal.

Among other papers giving proofs of correctness of garbage collection are the following. Dijkstra [3] presents an algorithm for a garbage collector running in parallel with the mutator, and proves it correct; Ben-Ari [1] improves on the algorithm and simplifies the proof. Shapiro [10] proves correct in an informal way a protocol for garbage collection in a fault-tolerant, distributed object-oriented system.

The communication model in our language is based on Milner's CCS [6]. A full description of CCS and its proof methods is [7]. Our calculus also has some similarities with the γ-calculus proposed by Boudol in [2]: his τ-actions, on which the evaluation of the language is based, corresponds to our unobservable transitions. However, the goal of his paper is to develop a calculus for communicating systems which has λ-calculus as a sub-calculus.

6 Summary and future work

A strength of our model is that, being high-level, the various correctness proofs can be given formally in a concise manner. However, the atomicity of the transitions in the examples provided is too coarse. This can be addressed within the model developed by allowing the transitions to be more fragmented; for example the communication transition may be split in its two components, sending and receiving, with a way of recording that a message is "in transit". Also, the garbage collection may be allowed to proceed in parallel with the computation even on the same node (as described in [3]). This can be achieved by raising the *scan* relation to a transition in the language, together with a way of synchronizing the mutator with the collector. However, the more refined the models are, the more complicated the proofs become.

An interesting extension of this work is to take into consideration various faults which can occur in distributed systems. To this aim, relations describing specific faults (e.g. a communication relation where the message is not received at the other node) can be added to the garbage collector relation. Proving that the algorithm implementing the garbage collector preserves bisimilarity even in the presence of these relations would be proof that the collector is resilient to these failures.

References

[1] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6, 1984.

[2] Gerard Boudol. Towards a lambda-calculus for concurrent and communicating systems. *TAPSOFT'89 LNCS 351*, pages 149–161, 1989.

[3] Edsger Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[4] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102, 1992.

[5] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. *Symposium on LISP and Functional Programming*, 1992.

[6] R. Milner. A calculus for communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[7] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[8] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Functional Programming Languages and Computer Architectures*, 1995.

[9] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[10] Mark Shapiro, David Plainfossé, and Olivier Gruber. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, INRIA, 1990.

A Proofs

Lemma 8 (Closed programs) $\forall a \in \mathbf{R}, E \xmapsto{a} F$ *and* $FV(E) = \emptyset \Rightarrow FV(F) = \emptyset$.
This is quite intuitive, since none of the transitions drops bindings in the heap, while those which add variables also add bindings for them. A formal proof can be given by transition induction.

Lemma 7 $(\xmapsto{fv})$ *is a strong bisimulation up to* $\simeq$.
Proof. *It will be enough to show:*

$$E \xmapsto{fv} F \text{ and } E \xmapsto{a} E' \Rightarrow \exists F', F'' \text{ s.t. } F \xmapsto{a} F' \text{ and } E' \xmapsto{fv} F'' \text{ and } F'' \mathcal{S}\ F',$$

where $\mathcal{S}$ is the substitution relation. We shall prove this by transition induction. Thus, we first show the property for transition whose deduction is done in one step (axioms) and then use the induction hypothesis to show the property for longer deductions.

Note that it order to show $E' \xmapsto{fv} F''$, it is enough to prove that $F'' \overset{p}{\approx} E'$ and $F'' \subseteq_H E'$. F'' is just an alphabetic variant of F', and F' closed follows from Lemma 8.

We will be implicitly using below the fact that whenever we have $E \overset{p}{\approx} F$ and $E \equiv \mathcal{E}[\langle\langle H_1, T\rangle\rangle]$, there exists $\mathcal{F}[] \overset{p}{\approx} \mathcal{E}[]$ s.t. $F \equiv \mathcal{F}[\langle\langle H_2, T\rangle\rangle]$.

Case 1: $E \xmapsto{a} E'$ is an axiom. We have to consider the following cases: $a = \alpha_v, \overline{\alpha}_v, if_0, if_1, sum_1, sum_2, comp$. This can only happen if $\exists \mathcal{E}[], H_1, T$ s.t. $E \equiv \mathcal{E}[\langle\langle H_1, T\rangle\rangle]$, and $\mathcal{E}[\langle\langle H_1, T\rangle\rangle] \xmapsto{a} E'$.

- *The derivation is an instance of $\overline{\alpha}_v$*
  $E \equiv \mathcal{E}[\langle\langle H_1, \alpha!x.T\rangle\rangle] \xmapsto{\overline{\alpha}_v} \mathcal{E}[\langle\langle H_1, T\rangle\rangle] \equiv E'$, *where* $v = Heap(E)(x)$.
  Because $E \xmapsto{fv} F$, we have $\mathcal{E}[\langle\langle H_1, \alpha!x.T\rangle\rangle] \overset{p}{\approx} F$ and $F \subseteq_H E$. The first of these implies that $\exists \mathcal{F}[], H_2.F \equiv \mathcal{F}[\langle\langle H_2, \alpha!x.T\rangle\rangle$ and $\mathcal{F}[] \overset{p}{\approx} \mathcal{E}[]$. Let $F'' \equiv F' \equiv \mathcal{F}[\langle\langle H_2, T\rangle\rangle]$.
  It is obvious that $F \xmapsto{\overline{\alpha}_v} F'$ since x must be bound to the same value. It remains to show $E' \xmapsto{fv} F'$.
  - $F' \subseteq_H E'$ is true by hypothesis.
  - $F' \overset{p}{\approx} E'$ is true because $\langle\langle H_1, T\rangle\rangle \overset{p}{\approx} \langle\langle H_2, T\rangle\rangle$ and $\mathcal{E}[] \overset{p}{\approx} \mathcal{F}[]$.

- α_v
  $E \equiv \mathcal{E}[\langle\langle H_1, \alpha?x.T\rangle\rangle] \xmapsto{fv} \mathcal{F}[\langle\langle H_2, \alpha?x.T\rangle\rangle] \equiv F$, *where* $\mathcal{E}[] \overset{p}{\approx} \mathcal{F}[]$ *and* $F \subseteq_H E$.
  $\mathcal{E}[\langle\langle H_1, \alpha?x.T\rangle\rangle] \xmapsto{\alpha_v} \mathcal{E}[\langle\langle H_1 \cup \{x' = v\}, T\{x'/x\}\rangle\rangle] \equiv E'$, *where* $x' \notin Dom(Heap(E))$.
  Let $F' \equiv \mathcal{F}[\langle\langle H_2 \cup \{x'' = v\}, T\{x''/x\}\rangle\rangle]$, *where* $x'' \notin Dom(Heap(F))$. *Obviously* $F \xmapsto{\alpha_v} F'$.
  Let $F'' \equiv F'\{x''/x'\}$. *It is easy to check that* $E' \xmapsto{fv} F''$.

- *If_0 (and similarly If_1, Sum_1, Sum_2)*
  $E \equiv \mathcal{E}[\langle\langle H_1, \textbf{if } x \textbf{ then } T_1 \textbf{ else } T_2\rangle\rangle] \xmapsto{if_0} \mathcal{E}[\langle\langle H_1, T_2\rangle\rangle]$, *because* $Heap(E)(x) = 0$.

$\mathcal{E}[\langle\!\langle H_1, \textbf{if } x \textbf{ then } T_1 \textbf{ else } T_2\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, \textbf{if } x \textbf{ then } T_1 \textbf{ else } T_2\rangle\!\rangle] \equiv F$, where $\mathcal{F}[] \overset{p}{\approx} \mathcal{E}[]$ and $F \subseteq_H E$. Let $F' \equiv F'' \equiv \mathcal{F}[\langle\!\langle H_2, T_2\rangle\!\rangle]$.

- We verify $F \xmapsto{if_0} F'$. Since F is closed we must have $x \in Dom(Heap(F))$.
But $Heap(F) \subseteq Heap(E)$, hence $Heap(F)(x) = Heap(E)(x)$.

- $E' \xmapsto{fv} F'$ is also true.

- *Comp*
  $E \equiv \mathcal{E}[\langle\!\langle H, \alpha!C[e].T\rangle\!\rangle] \xmapsto{exp} \mathcal{E}[\langle\!\langle H', \alpha!C[e'].T\rangle\!\rangle]$ because $(H, C[e], Heap(\mathcal{E}[])) \xmapsto{exp} (H', C[e'])$, where $exp \in \{alloc, app, assign\}$.
  On the other hand, $E \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, \alpha!C[e].T\rangle\!\rangle] \equiv F$, where $F \subseteq_H E$ and $\mathcal{F}[] \overset{p}{\approx} \mathcal{E}[]$.

  – *Alloc*
    $(H, C[h], Heap(\mathcal{E}[])) \xmapsto{alloc} (H \uplus \{x = h\}, C[x])$, where $x \notin Dom(Heap(E))$.
    Let $F' \equiv \mathcal{F}[\langle\!\langle H_2 \uplus \{x_2 = h\}, \alpha!C[x_2].T\rangle\!\rangle]$, and $F'' = F'\{x_2/x\}$. It can be seen that $F \xmapsto{comp} F'$ and $E' \xmapsto{fv} F''$.

  – *App, Assign are treated similarly.*

Case 2: the inference rules.

- *Com_1 (Com_2 is similar)*
  $E \equiv \mathcal{E}[\langle\!\langle H_1, T_1 \parallel T_2\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, T_1 \parallel T_2\rangle\!\rangle] \equiv F$
  E has one of the following actions: $a \in \{\alpha_v, \overline{\alpha}_v, if_0, if_1, sum_1, sum_2, comp, comm_l\}$.
  $\mathcal{E}[\langle\!\langle H_1, T_1 \parallel T_2\rangle\!\rangle] \xmapsto{a} \mathcal{E}[\langle\!\langle H_1', T_1' \parallel T_2\rangle\!\rangle]$ because $\mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle] \xmapsto{a} \mathcal{E}[\langle\!\langle H_1', T_1'\rangle\!\rangle]$.
  Now, consider $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle]$. We have $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle] \overset{p}{\approx} \mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle]$, $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle] \subseteq_H \mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle]$ by assumption, and $FV(\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle]) \subseteq FV(\mathcal{F}[\langle\!\langle H_2, T_1 \parallel T_2\rangle\!\rangle]) = \emptyset$ (by properties of free variables). So $\mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle]$.
  Since the proof of the transition $\mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle] \xmapsto{a} \mathcal{E}[\langle\!\langle H_1', T_1'\rangle\!\rangle]$ is shorter, using the i.h., we have that $\exists H_2'.\mathcal{E}[\langle\!\langle H_1', T_1'\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2', T_1'\rangle\!\rangle]$ and $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle] \xmapsto{a} \mathcal{F}[\langle\!\langle H_2', T_1'\rangle\!\rangle]$.
  We now consider $F' \equiv \mathcal{F}[\langle\!\langle H_2', T_1' \parallel T_2\rangle\!\rangle]$. We must show $\mathcal{E}[\langle\!\langle H_1', T_1' \parallel T_2\rangle\!\rangle] \xmapsto{fv} F'$:
  $\mathcal{E}[\langle\!\langle H_1', T_1' \parallel T_2\rangle\!\rangle] \overset{p}{\approx} \mathcal{F}[\langle\!\langle H_2', T_1' \parallel T_2\rangle\!\rangle]$ is obvious
  $\mathcal{E}[\langle\!\langle H_1', T_1' \parallel T_2\rangle\!\rangle] \subseteq_H \mathcal{F}[\langle\!\langle H_2', T_1' \parallel T_2\rangle\!\rangle]$ because $\mathcal{F}[\langle\!\langle H_2', T_1'\rangle\!\rangle] \subseteq_H \mathcal{E}[\langle\!\langle H_1', T_1'\rangle\!\rangle]$

- *Fix*
  $E \equiv \mathcal{E}[\langle\!\langle H_1, \textbf{fix } \{X = T\}\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, \textbf{fix } \{X = T\}\rangle\!\rangle] \equiv F$ and $E \xmapsto{a} \mathcal{E}[\langle\!\langle H_1', P\rangle\!\rangle]$.
  This transaction can only be inferred from $\mathcal{E}[\langle\!\langle H_1, T[\textbf{fix } \{X = T\}/X]\rangle\!\rangle] \xmapsto{a} \mathcal{E}[\langle\!\langle H_1', P\rangle\!\rangle]$.
  Consider $F\langle\!\langle H_2, T[\textbf{fix } \{X = T\}/X]\rangle\!\rangle$. We first show that:
  $\mathcal{E}[\langle\!\langle H_1, T[\textbf{fix } \{X = T\}/X]\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, T[\textbf{fix } \{X = T\}/X]\rangle\!\rangle]$. Using the i.h. $\exists H_2'$ s.t.
  $\mathcal{F}[\langle\!\langle H_2, T[\textbf{fix } \{X = T\}/X]\rangle\!\rangle] \xmapsto{a} \mathcal{F}[\langle\!\langle H_2', P\rangle\!\rangle]$ and $\mathcal{E}[\langle\!\langle H_1', P\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2', P\rangle\!\rangle]$. And this is the desired conclusion.

- *Comm*$_l$
  $E \equiv \mathcal{E}[\langle\!\langle H_1, T_1 \parallel T_2\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, T_1 \parallel T_2\rangle\!\rangle] \equiv F$
  Suppose that the comm$_l$ *transition is possible. Then, for some* α, v*:*
  $\mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle] \xmapsto{\overline{\alpha}_v} \mathcal{E}[\langle\!\langle H_1, T_1'\rangle\!\rangle]$ *and* $\mathcal{E}[\langle\!\langle H_1, T_2\rangle\!\rangle] \xmapsto{\alpha_v} \mathcal{E}[\langle\!\langle H_1', T_2'\rangle\!\rangle]$.
  Now take $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle]$. *Since* $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle] \overset{p}{\approx} \mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle]$ *and* $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle] \subseteq_H \mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle]$ *(by assumption), and* $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle]$ *closed* $(FV(T_1) \subseteq FV(T_1 \parallel T_2))$ *it follows that*
  $\mathcal{E}[\langle\!\langle H_1, T_1\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle]$. *Using the i.h.* $\mathcal{F}[\langle\!\langle H_2, T_1\rangle\!\rangle] \xmapsto{\overline{\alpha}_v} \mathcal{F}[\langle\!\langle H_2, T_1'\rangle\!\rangle]$ *and*
  $\mathcal{E}[\langle\!\langle H_1, T_1'\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2, T_1'\rangle\!\rangle]$. *Similarly, we obtain that*
  $\mathcal{E}[\langle\!\langle H_1', T_2'\rangle\!\rangle] \xmapsto{fv} \mathcal{F}[\langle\!\langle H_2', T_2'\rangle\!\rangle]$, *where* $\mathcal{F}[\langle\!\langle H_2, T_2\rangle\!\rangle] \xmapsto{\alpha_v} \mathcal{F}[\langle\!\langle H_2', T_2'\rangle\!\rangle]$.
  We then have $F \xmapsto{comm_l} F' \equiv \mathcal{F}[\langle\!\langle H_2', T_1' \parallel T_2'\rangle\!\rangle]$ *It can be verified that* $E' \xmapsto{fv} F'$ *is also true.*

- *Comm*$_r$
  $E \xmapsto{fv} F$ *and* $E \xmapsto{comm_r} E'$. *We have to show that* $\exists F'$ *s.t.* $F \xmapsto{comm_r} F'$ *and* $E' \xmapsto{fv} F'$.
  Now, $E \xmapsto{comm_r} E'$ *can only be inferred if* $E \equiv E_1 \odot E_2$, *and*
  $\exists v. E_1 \odot E_2 \xmapsto{\alpha_v} E_1' \odot E_2$, $E_1 \odot E_2 \xmapsto{\overline{\alpha}_v} E_1 \odot E_2'$ *and* $E' \equiv E_1' \odot E_2'$.
  Because $E \overset{p}{\approx} F$ *there must* $\exists F_1, F_2$ *s.t.* $F \equiv F_1 \odot F_2$ *with* $F_1 \overset{p}{\approx} E_1$ *and* $F_2 \overset{p}{\approx} E_2$.
  By i.h. (shorter inference) we have: $E_1 \odot E_2 \xmapsto{\alpha_v} E_1' \odot E_2$, $F_1 \odot F_2 \xmapsto{\alpha_v} F_1' \odot F_2$, *and*
  $E_1' \odot E_2 \xmapsto{fv} F_1' \odot F_2$. *Similarly,* $E_1 \odot E_2' \xmapsto{fv} F_1 \odot F_2'$. *It follows that* $E_1' \odot E_2' \xmapsto{fv} F_1' \odot F_2'$.
  Two things remain to be checked:
  - $E_1' \odot E_2' \overset{p}{\approx} F_1' \odot F_2'$ *because* $E_1' \overset{p}{\approx} F_1'$ *and* $E_2' \overset{p}{\approx} F_2'$.
  - $E_1' \odot E_2' \subseteq_H F_1' \odot F_2'$ *because* $Heap(F_1' \odot F_2') = Heap(F_1' \odot F_2)$ *and similarly for* E. *Using the i.h. the relation holds.*