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 [10]. 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 [9], 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.

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP '97 Amsterdam, ND
© 1997 ACM 0-89791-918-1/97/0006...$3.50

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 correctness of the garbage collector is very rarely proved in 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-**S** calculus of Felleisen and Hieb [4], and from Milner's CCS [9], 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. A *program* is a collection of one or more *processes* running in parallel on one or more nodes of a distributed system. A process consists of a *thread* and the local store it has access to, also called *heap*. The thread part of the process may contain sub-threads, all of which share the local store. The heap contains bindings. The semantics defined allows us to use many of the proof techniques developed for CCS, such as bisimulation and transition induction.

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 "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. Due to space limitations, many proofs have been omitted from this paper. A technical report including full proofs can be found at `http://cs.nyu.edu/csd/reports.html`.

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 one, it roughly corresponds to an impure functional language with threads in a distributed environment. Computation is distributed among a number of nodes, each node having a collection of local threads running in parallel and sharing a local heap. 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-S. Below, we will describe the syntax of the language, its semantics, and introduce the concept of observational bisimulation, on which we base the equivalence of programs.

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 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 store. The heap contains bindings which can be mutually recursive, since we allow modification of bindings through assignment. The syntax is presented in Figure 1; parentheses are necessary to disambiguate certain syntactic constructs.

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 perform 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; 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 well formed program does not contain recursive definitions with multiple solutions. If the solution is unique, it is denoted by fix $\{X = T\}$. We will precisely describe the behavior of all 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 uniqueness 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.

A variable may be bound to only a single value. Consequently, a heap can also be considered a finite function. $Dom(H)$ denotes the bound variables of H, and $Rng(H)$ denotes the values bound in H. Moreover, we require that a well formed program has the domains of all component heaps disjoint (i.e. 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*)	$x, y, z \in$	*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 $\lambda_{\parallel}$

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 rewrite 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. Note that, unlike in the sequential case, the execution step is not uniquely determined. Non-determinism can occur directly, as a result of the sum combinator, or indirectly as a result of parallel composition (more on this later). 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*

$$\begin{aligned} \mathit{Lbl} \;=\; & \{\alpha_v \mid \alpha \in \mathit{Port}, v \in \mathit{Hval}\} \cup \{\overline{\alpha}_v \mid \alpha \in \mathit{Port}, v \in \mathit{Hval}\} \\ & \cup \{\mathit{comp}, \mathit{if}_0, \mathit{if}_1, \mathit{sum}, \mathit{comm}_l, \mathit{comm}_r\} \end{aligned}$$

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*, written $\alpha_v \frown \overline{\alpha}_v$.

The definitions of the relations associated with each label are given inductively. The fact that a pair belongs to a relation will be asserted by some rules in the case the pair occurs as the conclusion of that rules. As a notational convention, we will denote the relations with names starting with lowercase letters. The rules have their first letter capitalized (with the exception of α_v and $\overline{\alpha}_v$). The distinction is necessary since a relation can be specified by more than one rule. Also, the "compatible closure" rules help define many relations. A rule can also have hypotheses, possibly requiring that some other pair of programs belongs to the same or different transition relation. Rules without hypotheses are called *axioms*. 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* [9].

As an example, the relation *sum* is defined by the rules *Sum*, *Fix*, Par_1 and Par_2. Rule *Sum* is an axiom, while rule Par_1 has a hypothesis. The rule Par_1 helps define not only *Sum* but also relations α_v, $\overline{\alpha}_v$, if_0, if_1, *comp* and comm_l.

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 (combinators) of that process.

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 v used in the label $\overline{\alpha}_v$ 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.

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 variable is defined for all the syntactic

Expression evaluation contexts, and instruction expression:

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

Evaluation rules for expressions:

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

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] & \xrightarrow{\overline{\alpha}_v} \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] & \xrightarrow{\alpha_h} \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, \text{if } x \text{ then } T \text{ else } U \rangle\!\rangle] & \xrightarrow{if_1} \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, \text{if } x \text{ then } T \text{ else } U \rangle\!\rangle] & \xrightarrow{if_0} \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] & \xrightarrow{sum} \mathcal{E}[\langle\!\langle H, T_i \rangle\!\rangle] & i = 1, 2 \\ (\mathit{Comp}) & \mathcal{E}[\langle\!\langle H, \alpha!C[e].T \rangle\!\rangle] & \xrightarrow{comp} \mathcal{E}[\langle\!\langle H', \alpha!C[e'].T \rangle\!\rangle] & (H, C[e], \mathit{Heap}(\mathcal{E}[\,])) \xrightarrow{exp} (H', C[e']), \\ & & & exp \in \{alloc, app, :=\} \\ (\mathit{Comm}_l) & \mathcal{E}[\langle\!\langle H, T \parallel U \rangle\!\rangle] & \xrightarrow{comm_l} \mathcal{E}[\langle\!\langle H', T' \parallel U' \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle] \xrightarrow{a} \mathcal{E}[\langle\!\langle H, T' \rangle\!\rangle] \\ & & & \mathcal{E}[\langle\!\langle H, U \rangle\!\rangle] \xrightarrow{b} \mathcal{E}[\langle\!\langle H', U' \rangle\!\rangle], a \frown b \\ (\mathit{Comm}_r) & \mathcal{E}[E \odot F] & \xrightarrow{comm_r} \mathcal{E}[E' \odot F'] & \mathcal{E}[E \odot F] \xrightarrow{a} \mathcal{E}[E' \odot F] \\ & & & \mathcal{E}[E \odot F] \xrightarrow{b} \mathcal{E}[E \odot F'], a \frown b \\ (\mathit{Fix}) & \mathcal{E}[\langle\!\langle H, \text{fix } \{Y = T\} \rangle\!\rangle] & \xrightarrow{a} \mathcal{E}[\langle\!\langle H', T' \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, T[\text{fix } \{Y = T\}/Y] \rangle\!\rangle] \xrightarrow{a} \mathcal{E}[\langle\!\langle H', T' \rangle\!\rangle] \\ (\mathit{Par}_1) & \mathcal{E}[\langle\!\langle H, T \parallel U \rangle\!\rangle] & \xrightarrow{a} \mathcal{E}[\langle\!\langle H', T' \parallel U \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, T \rangle\!\rangle] \xrightarrow{a} \mathcal{E}[\langle\!\langle H', T' \rangle\!\rangle] \\ (\mathit{Par}_2) & \mathcal{E}[\langle\!\langle H, T \parallel U \rangle\!\rangle] & \xrightarrow{a} \mathcal{E}[\langle\!\langle H', T \parallel U' \rangle\!\rangle] & \mathcal{E}[\langle\!\langle H, U \rangle\!\rangle] \xrightarrow{a} \mathcal{E}[\langle\!\langle H', U' \rangle\!\rangle] \end{array}$$

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

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

$$
\begin{array}{ll}
FV(x) & = \{x\} \\
FV(i) & = \emptyset \\
FV(\lambda x.e) & = FV(e)\backslash\{x\} \\
FV(e_1\ e_2) & = FV(e_1) \cup FV(e_2) \\
FV(x := e) & = \{x\} \cup FV(e)
\end{array}
$$

Sets of values

$$
\begin{array}{ll}
FV(\emptyset) & = \emptyset \\
FV(S \uplus \{v\}) & = FV(S) \cup FV(v)
\end{array}
$$

Heaps

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

Threads

$$
\begin{array}{ll}
FV(\varepsilon) & = \emptyset \\
FV(\alpha?x.T) & = FV(T)\backslash\{x\} \\
FV(\alpha!e.T) & = FV(e) \cup FV(T) \\
FV(X) & = \emptyset \\
FV(\text{fix } \{X = T\}) & = FV(T) \\
FV(T \parallel U) & = FV(T + U) = FV(T) \cup FV(U) \\
FV(\text{if } x \text{ then } T \text{ else } U) & = \{x\} \cup FV(T) \cup FV(U)
\end{array}
$$

Processes and Programs

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

Free process variables:

$$
\begin{array}{llll}
FP(\varepsilon) & = \emptyset & FP(X) & = \{X\} \\
FP(\alpha?x.T) & = FP(\alpha!e.T) = FP(T) & FP(\text{fix } \{X = T\}) & = FP(T)\backslash\{X\} \\
FP(T \parallel U) & = FP(T + U) = FP(T) \cup FP(U) & FP(\text{if } x \text{ then } T \text{ else } U) & = FP(T) \cup FP(U)
\end{array}
$$

Figure 5: Definition of FV and FP functions

components of programs (see Figure 7). 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. Rule *Sum* says that the thread $(T_1 + T_2)$ can behave (non-deterministically) as either T_1 or T_2.

Transition relation *comp* describes a computation step, whereby an expression is reduced to a value (in one or more steps); it is defined as the transitive closure of the union of relations *alloc*, *app* and :=. These sub-relations, which describe sequential computation, correspond to the reduction rules in λ_v-S. They are modified to account for the possibility of non-local bindings of variables.

The computation relation 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 [10], 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 into 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 (:=), the value of y replaces the value x is bound to.

There are also "compatible closure" rules: *Fix*, Par_1 and Par_2. The two *Par* 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.

Rules $Comm_l$ and $Comm_r$ specify how two threads capable of transitions with complementary labels (sending and receiving a value v at some port α) communicate with each other. Rule $Comm_l$ describes communication local to a node. Note that the local heap is modified. 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

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*, if_0, if_1, $comm_l$, $comm_r$ which describe how the computation can proceed without such interaction. We will use the following notation:

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

$\mathbf{R}$ 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 $\overline{\alpha}_v$ to define the evaluation relation will be given after we introduce observational simulation.

Perhaps an example would help clarify how evaluation is done. The program in Figure 6 consists of two processes. The left one is recursive; it is capable of continually receiving a value at port α and sending to port β the result of

now, the first is evaluating the argument of the application (an *alloc* step which results in the allocation of 0 onto the heap) and then an *app* step. The formal parameter x of the function $\lambda x.1$ is bound (upon renaming) to the actual argument h', and the body of the function replaces the application expression. Note that at the end of the steps described, both heaps contain only garbage.

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 observationally 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

ERROR: ioerror
OFFENDING COMMAND: image

STACK:

-filestream-
-filestream-
-mark-
-savelevel-
-savelevel-