Abstract Machines for Memory Management

Christopher Walton

Laboratory for Foundations of Computer Science
Division of Informatics
The University of Edinburgh
cdw@dcs.ed.ac.uk

June 17, 1999

Abstract

In composing the Definition of Standard ML [1], the authors chose not to give an account of the operation of memory management. This was the right decision when focussing upon the abstract description of a sophisticated high-level language. However, the issues associated with memory management cannot be ignored by the compiler writer who must make decisions which have a great impact on the performance of the compiled code.

In the first half of this report, an abstract machine formalism of an efficient tag-free garbage-collector is presented which exposes many important memory management details such at the heaps, stacks, and environments. This model provides an implementor with an unambiguous and precise description of a memory management strategy for Standard ML. The second half of the report extends the abstract machine model to provide a formalism of distributed memory management in the LEMMA interface [2]. The extended model provides an appropriate setting for an implementation of Concurrent ML [3].

Keywords: Memory Management; Abstract Machines; Concurrent ML

1 Introduction

A significant advantage of high-level programming languages, such as Standard ML, is that the programmer is freed from the error-prone tasks associated with memory management. This is reflected in *The Definition of Standard ML* [1] which avoids an explicit treatment of memory management except to say that "there are no [semantic] rules concerning the disposal of inaccessible addresses".

For many purposes, such as reasoning about programs, this approach is entirely suitable. Nonetheless, the issues of memory management cannot be entirely overlooked. An important use of the Definition is to serve as a guide for the compiler writer who must make memory management decisions that critically affect the performance of compiled code. Other authors have also argued for the usefulness of a semantic model in making precise the important notions of space-safety and data sharing [4].

In a previous paper [5] a novel abstract machine, with an explicit treatment of memory management, was presented for the purpose of describing a run-time code-replacement operation. The intention of this report is to extend this model to provide a complete semantic account of memory management in both a sequential and distributed setting.

Memory is typically modelled using a *stack* and a *heap*. The stack holds temporary values of known size whose lifetime is determined by function applications, e.g. function parameters and local variables. The heap holds all other values, e.g. closures and dynamic data structures. Memory management of the stack is relatively straightforward as values are simply added or removed from the top of the stack. By contrast, memory management of the heap is considerably more challenging.

Without a system for automatic memory management, the programmer is left to manage the heap using explicit allocation and de-allocation facilities, e.g. `malloc` and `free` in C. For non-trivial programs this can be a very significant burden as it is, in general, very hard to ensure that an area of memory will not be required by a later computation. If memory is de-allocated too hastily, then the program will fail when it requires a value that has been removed. Conversely, if memory is de-allocated too conservatively, then the program may exhaust the supply of memory available.

The prevailing technique for automatic memory management of the heap is garbage collection. Heap allocation is performed (implicitly) by the programmer, and de-allocation by the garbage collector. Garbage collection is based on the idea that if a value is reachable, either from the stack or from a set of roots, then it must not be discarded. A survey of different garbage collection techniques is presented in [6].

It is also worth noting that an alternative, called region based memory management, has recently been developed [7]. In this scheme, the memory is modelled using multiple stacks each containing values of a single type. A sophisticated region inference algorithm is used to determine the memory requirements of the program at compile-time, thereby avoiding the need for run-time garbage collection. Although this technique appears to be the ideal solution for memory management, it will not be used in this report as the inference algorithm is primarily designed for sequential operation and contains no obvious method of integration into a distributed environment.

The semantic account of sequential memory management in the first half of this report consists of a formalisation of the two-space copying garbage collection technique which extends the *tag-free* algorithm presented in [8]. The advantages of the two-space technique are well known, for example, the data is compacted during collection which improves locality, and cyclic garbage is removed. However, the primary reason for selecting this algorithm is that it extends naturally into a distributed setting.

In the distributed case, it is not enough to simply formalise the garbage collection operation. A number of additional operations are also required for distributed memory management, e.g. data sharing between processors. These operations are presented in the LEMMA interface definition [2]. Thus, the second-half of this report will concentrate on a formalisation of this interface.

The LEMMA memory interface was created to support a distributed concurrent extension of Standard ML [3]. In that extension, the language is augmented with a number of constructs for communication and concurrency (Figure 1).

$$
\begin{array}{lcl}
\texttt{channel} & : & \forall\,\alpha\,.\ \texttt{unit} \rightarrow \alpha\ \texttt{channel} \\
\texttt{send} & : & \forall\,\alpha\,.\ \alpha\ \texttt{channel} \times \alpha \rightarrow \texttt{unit} \\
\texttt{receive} & : & \forall\,\alpha\,.\ \alpha\ \texttt{channel} \rightarrow \alpha \\
\texttt{fork} & : & (\texttt{unit} \rightarrow \texttt{unit}) \rightarrow \texttt{unit} \\
\texttt{rfork} & : & \texttt{int} \times (\texttt{unit} \rightarrow \texttt{unit}) \rightarrow \texttt{unit}
\end{array}
$$

Figure 1: Concurrency Primitives.

Communication channels are created using the `channel` primitive. A thread sends a value to another thread using the `send` primitive, which takes two arguments: a channel and a value. The receiving thread uses the `receive` primitive, which takes a channel argument. Both threads are blocked until the value is passed, which happens atomically. Channels are typed, that is the type system ensures that values sent and received on a particular channel have the same type. Threads are created either using the `fork` primitive, which runs the child thread on the same processor as its parent, or the `rfork` primitive, which runs the child process on a specified processor. Both `fork` and `rfork` take a function argument, whose body is evaluated in the child thread. When the function returns, the thread is terminated. The `rfork` primitive takes an additional argument which specifies where the child thread is to run. An example illustrating the concurrency primitives is shown in Figure 2. In this example, a thread is created on processor 2 that simply waits for an integer value on channel `c`, then returns the value, incremented by 3, on the same channel. The example concludes by sending the value 7 to the thread, and returns the response (which will be 10).

```
let val c = channel()
    val _ = rfork(2, fn _ => send(c, receive(c) + 3))
    val _ = send(c, 7)
in
    receive(c)
end
```

Figure 2: Concurrent ML Example.

2 The $\mathcal{M}\Lambda$ Language

Throughout this report, the formalisation of memory management is illustrated with respect to a typed call-by-value lambda language. The language is representative of a typical intermediate language used in a modern ML compiler. By basing the formalisation on such an intermediate language, it is possible to demonstrate that the memory-management operations are applicable to the whole of Standard ML (with the exception on the module system) without encountering a great deal of the complexity. For example, Standard ML pattern matching appears as simpler switch statements in the intermediate language, having been converted by a higher-level match compiler.

$$
\begin{array}{llcl}
\text{Type} & \tau & ::= & tn \mid tn(\tau) \mid \overline{\tau}^{\,k} \mid \tau_1 \to \tau_2 \mid tv \\
\text{Type Scheme} & \sigma & ::= & \tau \mid \forall\, \overline{tv}^{\,k}.\ \tau \\
& & & \\
\text{Program} & P & ::= & (\overline{\overline{D}},\ \overline{\overline{X}},\ \Lambda) \\
\text{Datatype} & D & ::= & \mathbf{datatype}\ tn\ \mathbf{of}\ \overline{(con,\ \tau)}^{\,k} \\
& & \mid & \mathbf{datatype}\ (\overline{tv}^{\,k},\ tn)\ \mathbf{of}\ \overline{(con,\ \tau)}^{\,l} \\
\text{Exception} & X & ::= & \mathbf{exception}\ (con,\ \tau) \\
\text{Expression} & \Lambda & ::= & \mathbf{scon}\ scon \\
& & \mid & \mathbf{con}\ con \mid \mathbf{con}\ (con,\ \Lambda) \\
& & \mid & \mathbf{con}\ (con,\ \overline{\tau}^{\,k}) \mid \mathbf{con}\ (con,\ \overline{\tau}^{\,k},\ \Lambda) \\
& & \mid & \mathbf{decon}\ (con,\ \Lambda) \mid \mathbf{decon}\ (con,\ \overline{\tau}^{\,k},\ \Lambda) \\
& & \mid & \mathbf{assign}\ (\Lambda_1,\ \Lambda_2) \\
& & \mid & \mathbf{tuple}\ \overline{\Lambda}^{\,k} \mid \mathbf{select}\ (i,\ \Lambda) \\
& & \mid & \mathbf{prim}\ (op,\ \overline{\Lambda}^{\,k}) \\
& & \mid & \mathbf{var}\ lv \mid \mathbf{var}\ (lv,\ \overline{\tau}^{\,k}) \\
& & \mid & \mathbf{let}\ (lv,\ \sigma) = \Lambda_1\ \mathbf{in}\ \Lambda_2 \mid \mathbf{let}\ \overline{(lv,\ \sigma)}^{\,k} = \Lambda_1\ \mathbf{in}\ \Lambda_2 \\
& & \mid & \mathbf{fix}\ \overline{(lv,\ \sigma) = \Lambda_1}^{\,k}\ \mathbf{in}\ \Lambda_2 \\
& & \mid & \mathbf{fn}\ (lv,\ \tau_1 \to \tau_2) = \Lambda \mid \mathbf{fn}\ (\overline{lv}^{\,k},\ \overline{\tau_1}^{\,k} \to \tau_2) = \Lambda \\
& & \mid & \mathbf{app}\ (\Lambda_1,\ \Lambda_2) \mid \mathbf{app}\ (\Lambda_1,\ \overline{\Lambda_2}^{\,k}) \\
& & \mid & \mathbf{switch}\ \Lambda_1\ \mathbf{case}\ (c \overset{map}{\mapsto} \Lambda_2,\ \Lambda_3) \\
& & \mid & \mathbf{raise}\ (\Lambda,\ \tau) \mid \mathbf{handle}\ \Lambda_1\ \mathbf{with}\ \Lambda_2 \\
& & \mid & \mathbf{fork}\ \Lambda \mid \mathbf{rfork}\ (\Lambda_1,\ \Lambda_2) \\
& & \mid & \mathbf{channel}\ () \\
& & \mid & \mathbf{send}\ (\Lambda_1,\ \Lambda_2) \mid \mathbf{receive}\ \Lambda
\end{array}
$$

Figure 3: $\mathcal{M}\Lambda$ Abstract Syntax.

Notation: A set is defined by enumerating its members in braces, such as $\overline{\overline{x}} = \{a,\ b,\ c\}$ with $\emptyset$ for the empty set. A sequence is an ordered list of members of a set, e.g. $\overline{x}^{\,k} = (a,\ b,\ c,\ a)$. The ith element of a non-empty sequence is written x^i, where $0 < i \leq k$. A finite map from $\overline{x}^{\,k}$ to $\overline{y}^{\,k}$ is defined: $x \overset{map}{\mapsto} y = \{x^1 \mapsto y^1,\ \ldots,\ x^k \mapsto y^k\}$ (the elements of $\overline{x}^{\,k}$ must be unique). The domain Dom and range Rng are the sets of elements of $\overline{x}^{\,k}$ and $\overline{y}^{\,k}$ respectively. A stack is written as a dotted sequence, e.g. $S = (a{\cdot}b{\cdot}c)$. The left-most element of the sequence is the top of the stack, and a pair of adjacent brackets () is used to represent the empty stack.

The syntax of the $\mathcal{M}\Lambda$ language is shown in Figure 3. The syntactic categories of the language include special constants *scon* (unit, integer, word, real, character, and string) and constructors *con* such as c_true and e_match, with c ranging over both of these, and i over special constants of integer type. The meta-variables *tn*, *tv*, and *lv* are used for type names, type variables, and lambda variables. Type variables *tv* are bound to types, and lambda variables *lv* are bound uniquely to values generated by the evaluation of expressions.

The types τ are either constructor types (which may be nullary or unary), tuple types, functional types, or type variables. Expressions are provided for creating values of each of these types directly, with the exception of type variables. Constructor types *tn* and *tn*(τ) include the basic types, as required by the special constants; value constructor types; reference constructor types; and exception constructor types. Type-schemes σ can represent *polymorphic* types. For example, the polymorphic identity function is represented by the type scheme $\forall\, \alpha \,.\, \alpha \rightarrow \alpha$, where α is a type variable.

A program in the language contains a set of datatype declarations, a set of exception declarations, and an expression. A datatype declaration consists of a unique type name and a list of typed constructors. Each exception declaration consists of a unique exception name and an exception type. The expressions divide into those for constructing and de-constructing values, defining and manipulating variables, and controlling the order of evaluation. A small example is shown in Figure 4 to illustrate the differences between Standard ML and $\mathcal{M}\Lambda$.

Standard ML:

```
exception Factorial
fun fac n = if (n < 0) then raise Factorial
            else if (n = 0) then 1
            else n * fac(n - 1)
fac 10;
```

$\mathcal{M}\Lambda$ **Translation:**

(∅, **exception** (e_factorial, t_unit),
    fix (fac, t_int → t_int) =
        fn (n, t_int → t_int) =
            switch (**prim** (LT_i, (**var** n, **scon** 0)))
            case ({c_true ↦ **raise** (**con** e_factorial, t_int),
                c_false ↦
                    switch (**prim** (EQ_i, (**var** n, **scon** 0)))
                    case ({c_true ↦ **scon** 1,
                        c_false ↦
                            prim (MUL_i, (**var** n,
                                app (**var** fac, **prim** (SUB_i, (**var** n, **scon** 1)))))},
                    raise (**con** e_match, t_int))},
            raise (**con** e_match, t_int))
    in app (**var** *fac*, **scon** 10))

Figure 4: Factorial Example.

3 Sequential Memory Management

In order to formalise the sequential garbage collection operation, it is necessary to define the dynamic semantics of the sequential sub-language (i.e. excluding the communication and concurrency operations). The dynamic semantics of $\mathcal{M}\Lambda$ are formalised by a transition relation between states of an abstract machine. The machine describes the memory management behaviour of an implementation of the language, except that it abstracts from the allocation of environments. The allocation of environments simply adds extra baggage to the rules, for a treatment of this topic see [9]. The organisation of the $\mathcal{M}\Lambda$ abstract machine has some features in common with the $\lambda_{\mathrm{gc}}^{\to\forall}$ abstract machine [4] which is used in the formal description of the behaviour of the TIL/ML compiler. However, the transitions differ considerably as $\mathcal{M}\Lambda$ does not adopt the named-form representation of expressions and types.

The syntax of the abstract machine is given in Figure 5. The state of the machine is defined by a 4-tuple $(H,\ E,\ ES,\ RS)$ of a heap, an environment, an exception stack, and a result stack. The heap is used to store all the run-time data of the program, while the environment provides a view of the heap relevant to the fragment of the program being evaluated, e.g. a mapping between the bound variables currently in scope, and their values on the heap. The exception stack stores pointers to exception handling functions (closures). The result stack holds pointers to temporary results during evaluation. Thus, the *memory* of the machine is effectively modelled by the heap, as the stacks simply contain pointers into the heap.

Machine State	M	$::=$	$(H,\ E,\ ES,\ RS)$
Heap	H	$::=$	$(TH,\ VH)$
Type Heap	TH	$::=$	$p \overset{map}{\mapsto} ty$
Heap Types	ty	$::=$	$tn \mid tn(p) \mid$ $\overline{p}^{\,k} \mid p_1 \to p_2 \mid$ $tv \mid \forall\, \overline{p_1}^{\,k} \,.\, p_2$
Value Heap	VH	$::=$	$l \overset{map}{\mapsto} val$
Heap Values	val	$::=$	$scon \mid con \mid con(l) \mid$ $\overline{l}^{\,k} \mid \langle\!\langle E,\ \overline{lv}^{\,k},\ \Lambda \rangle\!\rangle \mid \Omega$
Environment	E	$::=$	$(TE,\ CE,\ VE)$
Type Env.	TE	$::=$	$\overline{\overline{tn}}$
Constructor Env.	CE	$::=$	$con \overset{map}{\mapsto} p$
Variable Env.	VE	$::=$	$lv \overset{map}{\mapsto} (l,\ p)$
Exception Stack	ES	$::=$	$() \mid (l,\ p){\cdot}ES$
Result Stack	RS	$::=$	$() \mid p{\cdot}RS \mid (l,\ p){\cdot}RS \mid E{\cdot}RS$

Figure 5: $\mathcal{M}\Lambda$ Abstract Machine Syntax.

The heap consists of a type-heap mapping pointers to allocated types, and a value-heap mapping locations to allocated values. The heap types correspond directly to types in the $\mathcal{M}\Lambda$ language, and the heap values correspond to the heap types. Nullary constructors *scon* and *con* have type *tn*. Unary constructors $con(l)$ have type $tn(p)$. Tuples $\overline{l}^{\,k}$ have type $\overline{p}^{\,k}$ and function closures $\langle\!\langle E,\ \overline{lv}^{\,k},\ \Lambda\rangle\!\rangle$ have type $p_1 \rightarrow p_2$. The type heap and value heap are represented by finite-maps, as heap-locations and type-pointers may be bound only once.

The *shape* of the data at a particular heap-location is determined by its corresponding type. Without this type information, a heap value is simply a collection of binary information. The type information makes clear the internal representations of the value, e.g. pointers for garbage collection. Thus, each heap-location must be paired with a type-pointer: $(l,\ p)$.

It is important to note that there is no explicit notion of a memory *address*. For example, it is not possible to perform pointer arithmetic, e.g. $p_1 + p_2$. The only operations permitted on type-pointers and heap-locations are the comparison for equality, e.g. $p_1 = p_2$, and the retrieval of the corresponding value from the heap, e.g. $H(p) = tn$.

The following syntactic conventions are used for performing heap allocations: $H[l_1 \mapsto val_1,\ \ldots,\ l_k \mapsto val_k]$ allocates values $val_1,\ \ldots,\ val_k$ on the value heap, binding them to fresh locations $l_1, \ldots, l_k$. $H[p_1 \mapsto \tau_1,\ \ldots,\ p_k \mapsto \tau_k]$ allocates types $\tau_1,\ \ldots,\ \tau_k$ on the type heap, binding them to fresh type-pointers $p_1,\ \ldots,\ p_k$. There are no corresponding operations for removing values or types from the heap as this is achieved through garbage collection. However, the assignment of references and the fixed-point operator require a heap-update operation. Assignment uses the update operation $H[l_1 \overset{upd}{\mapsto} \mathrm{c\_ref}(l_2)]$ to update the reference at l_1 to $\mathrm{c\_ref}(l_2)$. This is clearly a trivial operation as it only requires the update of a single heap-location. The fixed-point case is slightly more complex: $H[l \mapsto \Omega]$ allocates a dummy closure on the value heap bound to a fresh heap-location l. This location can subsequently be updated with a mapping to a closure $H[l \overset{upd}{\mapsto} \langle\!\langle E,\ \overline{lv}^{\,k},\ \Lambda\rangle\!\rangle]$. In this case, a suitably-sized area of the heap must be reserved to hold the closure.

$$
\begin{array}{lcl}
M & = & (H,\ E,\ (),\ (),\ P) \\[1ex]
H & = & (TH,\ \emptyset) \\
TH & = & \{p_1 \mapsto \mathrm{t\_unit} \rightarrow \mathrm{t\_bool}, \\
 & & \ p_2 \mapsto tv^1, \quad p_3 \mapsto \forall\ (p_2)\ .\ \mathrm{t\_unit} \rightarrow \mathrm{t\_list}(p_2), \\
 & & \ p_4 \mapsto tv^2, \quad p_5 \mapsto \forall\ (p_4)\ .\ (p_4,\ \mathrm{t\_list}(p_4)) \rightarrow \mathrm{t\_list}(p_4), \\
 & & \ p_6 \mapsto tv^3, \quad p_7 \mapsto \forall\ (p_6)\ .\ p_6 \rightarrow \mathrm{t\_ref}(p_6), \\
 & & \ p_8 \mapsto \mathrm{t\_unit} \rightarrow \mathrm{t\_exn}\} \\[1ex]
E & = & (TE,\ CE,\ \emptyset) \\
TE & = & \{\mathrm{t\_unit},\ \mathrm{t\_int},\ \mathrm{t\_word},\ \mathrm{t\_real},\ \mathrm{t\_char},\ \mathrm{t\_string},\ \mathrm{t\_bool}, \\
 & & \ \mathrm{t\_list},\ \mathrm{t\_ref},\ \mathrm{t\_exn}\} \\
CE & = & \{\mathrm{c\_true} \mapsto p_1,\ \mathrm{c\_false} \mapsto p_1, \\
 & & \ \mathrm{c\_nil} \mapsto p_3,\ \mathrm{c\_cons} \mapsto p_5,\ \mathrm{c\_ref} \mapsto p_7, \\
 & & \ \mathrm{e\_match} \mapsto p_8,\ \mathrm{e\_bind} \mapsto p_8\}
\end{array}
$$

Figure 6: Initial Machine State.

The environment records the allocation of $\mathcal{M}\Lambda$ values, mapping them to pairs of heap locations and type-pointers. As identifiers and variables are unique, their corresponding environments are represented by finite-maps, with the exception of the type environment where it is sufficient just to use a set for type names. The following notational conventions are used for extending the environment: $E[tn]$ adds tn to the type environment, $E[con \mapsto p]$ binds the constructor con to the heap pointer p in the constructor environment. Similarly, $E[lv \mapsto (l,\ p)]$ denotes the binding of a lambda variable to a type-pointers and heap-location in the environment. There are no operations for removing bindings from the environment. However, unlike the heap, a copy of the current environment may be made at any time, for example by creating a closure. Thus, bindings can effectively be removed from the environment by reverting to an old copy of the environment.

Execution of the abstract machine is defined by a transition system between machine states. The individual transitions are listed in Appendix A. The top-level transition has the form $(H_1,\ E_1,\ ES_1,\ RS_1,\ P) \Rightarrow (H_2,\ E_2,\ ES_2,\ RS_2)$, where P is an $\mathcal{M}\Lambda$ program, $(H_1,\ E_1,\ ES_1,\ RS_1)$ is the initial machine state (as illustrated in Figure 6), and $(H_2,\ E_2,\ ES_2,\ RS_2)$ is the final machine state.

The majority of the rules in Appendix A concern the evaluation of expressions: $(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l,\ p){\cdot}RS_2)$. Evaluating an expression always generates a new machine state and leaves a single pair on the result stack. A new machine state is generated at each step as garbage collection may occur in the middle of an evaluation (this will be discussed in the next section). The transitions for the **prim** expression do not appear as they are largely trivial. The primitive operations, as defined in the Standard ML Initial Basis, are listed in Figure 7.

There are three possible outcomes which result from an evaluation: termination, exceptional termination and non-termination. Firstly, the sequence of transitions may terminate normally yielding a single pair $(l,\ p)$ in the result stack which references the result. Secondly, the sequence may terminate prematurely, through an uncaught exception, yielding a pair $(l,\ p)$ at the top of the result stack which references the exception. Finally, the machine may encounter an infinite sequence of transitions and fail to terminate.

Absolute Value (`abs`)	ABS_i, ABS_r
Negate (`~`)	NEG_i, NEG_r
Divide (`/`)	DIV_r
Integer Divide (`div`)	DIV_i, DIV_w
Modulo (`mod`)	MOD_i, MOD_w
Multiply (`*`)	MUL_i, MUL_w, MUL_r
Add (`+`)	ADD_i, ADD_w, ADD_r
Subtract (`-`)	SUB_i, SUB_w, SUB_r
Less (`<`)	LT_i, LT_w, LT_r, LT_c, LT_s
Greater (`>`)	GT_i, GT_w, GT_r, GT_c, GT_s
Equal (`=`)	EQ_i, EQ_w, EQ_r, EQ_c, EQ_s

i = integer, w = word, r = real, c = char, s = string

Figure 7: Primitive Operations (*op*).

A number of the evaluation rules in Appendix A (e.g. Rule 85) make use of the $instance(p_\sigma,\ p_\tau)$ operation which performs the instantiation of the type-scheme referenced by p_σ with the types referenced by p_τ (i.e. the type variables in p_σ are substituted with types from p_τ). The rules associated with this operation appear in Figure 8. These rules Rule 1 creates a substitution environment $\mathcal{S}$ from the type variables of p_σ to the types of p_τ. The $subst(\mathcal{S},\ p)$ operation (Rules 2 to 7) are then invoked to perform the substitution. There is a separate case for each of the types in the language. Substitution is performed recursively until all if the types have been examined. Rule 7 performs the actual substitution of a type variable with a type from $\mathcal{S}$.

$$\frac{\begin{array}{l} H_1(p_\sigma) = \forall\ \overline{p_1}^{\,k}\ .\ p_2 \\ H_1(p_\tau) = \overline{p_3}^{\,k} \\ \mathcal{S} = \{p_1^1 \mapsto p_3^1,\ \ldots,\ p_1^k \mapsto p_3^k\} \\ (H_1,\ E,\ ES,\ RS_1,\ subst(\mathcal{S},\ p_2)) \Rightarrow (H_2,\ E,\ ES,\ RS_2) \end{array}}{(H_1,\ E,\ ES,\ RS_1,\ instance(p_\sigma,\ p_\tau)) \Rightarrow (H_2,\ E,\ ES,\ RS_2)} \tag{1}$$

$$\frac{H(p) = tn}{(H,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p)) \Rightarrow (H,\ E,\ ES,\ p{\cdot}RS)} \tag{2}$$

$$\frac{\begin{array}{l} H_1(p_1) = tn(p_2) \\ (H_1,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_2)) \Rightarrow (H_2,\ E,\ ES,\ p_3{\cdot}RS) \end{array}}{(H_1,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_1)) \Rightarrow (H_2[p_4 \mapsto tn(p_3)],\ E,\ ES,\ p_4{\cdot}RS)} \tag{3}$$

$$\frac{\begin{array}{l} H_1(p_1) = \overline{p_2}^{\,k} \\ (H_1,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_2^1)) \Rightarrow (H_2,\ E,\ ES,\ p_3^1{\cdot}RS)\ \cdots \\ \quad (H_k,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_2^k)) \Rightarrow (H_{k+1},\ E,\ ES,\ p_3^k{\cdot}RS) \end{array}}{(H_1,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_1)) \Rightarrow (H_{k+1}[p_4 \mapsto \overline{p_3}^{\,k}],\ E,\ ES,\ p_4{\cdot}RS)} \tag{4}$$

$$\frac{\begin{array}{l} H_1(p_1) = p_2 \to p_3 \\ (H_1,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_2)) \Rightarrow (H_2,\ E,\ ES,\ p_4{\cdot}RS) \\ (H_2,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_3)) \Rightarrow (H_3,\ E,\ ES,\ p_5{\cdot}RS) \end{array}}{(H_1,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p_1)) \Rightarrow (H_3[p_6 \mapsto p_4 \to p_5],\ E,\ ES,\ p_6{\cdot}RS)} \tag{5}$$

$$\frac{H(p) = tv \qquad p \notin \mathrm{Dom}\ \mathcal{S}}{(H,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p)) \Rightarrow (H,\ E,\ ES,\ p{\cdot}RS)} \tag{6}$$

$$\frac{H(p) = tv \qquad p \in \mathrm{Dom}\ \mathcal{S}}{(H,\ E,\ ES,\ RS,\ subst(\mathcal{S},\ p)) \Rightarrow (H,\ E,\ ES,\ \mathcal{S}(p){\cdot}RS)} \tag{7}$$

Figure 8: Polymorphic Type Instantiation.

3.1 Sequential Garbage Collection

The following semantic account of sequential memory management consists of a formalisation of two-space copying garbage collection. Before proceeding, it is necessary to obtain an understanding of the basic algorithm. The address space of the heap is divided into two contiguous *semi-spaces*. During normal program execution, only one of these semi-spaces is used. Memory is allocated in a linear fashion until garbage collection appears to be profitable. At this point, the copying collector is called to reclaim space. The current semi-space (*from* space) is recursively scanned from the root objects, and all reachable objects are copied into the other semi-space (*to* space). When all of the objects that are reachable from the roots have been copied, the collection is finished, and the old semi-space (*from* space) can be discarded (see Figure 9). Subsequent memory allocations are performed in the new space (*to* space). The role of the semi-spaces is then reversed for the next garbage collection.

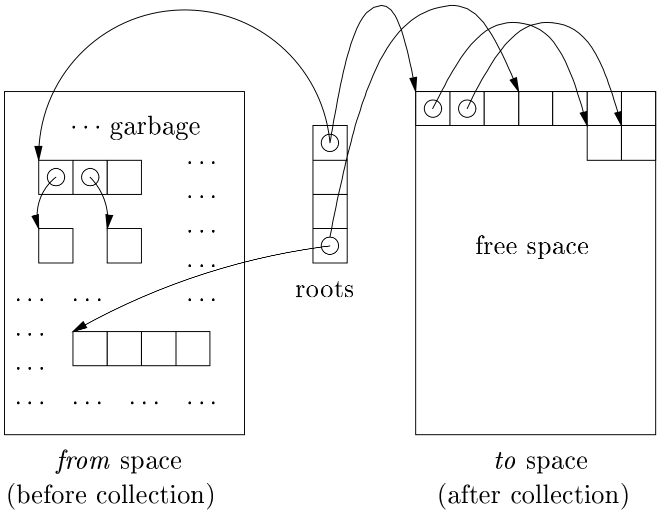

Figure 9: Two-Space Copying Garbage Collection

The $\mathcal{M}\Lambda$ garbage collection algorithm uses the type-based, tag-free style of [8]. The key idea is to preserve all heap values and types that are reachable either directly or indirectly from the current environment and stacks. The type information is used to determine the shape of the heap values. This allows the extraction of further locations and their types from the heap values without resorting to extra tags on the values themselves.

The garbage collector uses several data structures illustrated in Figure 10. The state of the garbage collector is the 4-tuple (Hf, Ht, PF, LF). The *from* space (heap) is denoted Hf, and the *to* space (heap) is denoted Ht. When a type or a value is copied from Hf into Ht, an entry is created in either PF or LF respectively. An entry $p_1 \mapsto p_2$ in PF represents a copied type where p_1 is the old type-pointer in Hf, and p_2 is the new type-pointer in Ht. Similarly, an entry $l_1 \mapsto l_2$ in LF represents a copied value. These forwarding tables are used during garbage collection to ensure that a type or value is copied only once between spaces.

Garbage Collection State	GC	::=	$(Hf,\ Ht,\ PF,\ LF)$
From Semi-space	Hf	::=	H
To Semi-space	Ht	::=	H
Forwarding Type Pointer	PF	::=	$p_1 \mapsto p_2$
Forwarding Value Pointer	LF	::=	$l_1 \mapsto l_2$

Figure 10: Garbage Collection Structures.

The garbage collection operation is defined by a transition system between garbage collection states. The roots of the garbage collection are the current environment E, the exception stack ES, and the result stack RS. Garbage collection is incorporated into the dynamic semantics of $\mathcal{M}\Lambda$ through the rule in Figure 11. Garbage collection can occur when an expression Λ is to be evaluated. At this point, the machine is interrupted, the garbage collector is initialised, and collection proceeds from each of the roots in turn. Once collection has completed, the evaluation of Λ resumes with the new abstract machine state.

The complex issue of when to initiate a collection will not be dealt with in detail here. However, a discussion can be found in [10] where the authors arrive at the following rule:

- For some constant $R > 1$, when current memory usage is more than R times the amount of reachable data preserved by the previous garbage collection, start a new garbage collection.

The $\mathcal{M}\Lambda$ abstract machine transitions in Appendix A have been designed with garbage collection in mind. As stated earlier in Section 3, a new machine state is generated by the evaluation of each expression to reflect the fact that garbage collection may have occurred. A number of the other rules also involve extra garbage collection considerations. For example, in the **let** expression (Rule 98) a copy of the environment is placed on the result stack while the body of the let expression is evaluated. This environment is later restored from the result stack to remove any bindings created during the evaluation of the body. The numbering of these environments reflects the fact that collections may have occured during the evaluation, and the stored environment on the result stack may have been updated.

$$\frac{\begin{array}{l}(Hf,\ \emptyset,\ \emptyset,\ \emptyset,\ E_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ E_2)\\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ ES_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ ES_2)\\ (Hf,\ Ht_2,\ PF_2,\ LF_2,\ RS_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_3,\ RS_2)\\ (Ht_3,\ E_2,\ ES_2,\ RS_2,\ \Lambda) \Rightarrow (Ht_4,\ E_3,\ ES_3,\ RS_3)\end{array}}{(Hf,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (Ht_4,\ E_3,\ ES_3,\ RS_3)} \tag{8}$$

Figure 11: Garbage Collection Introduction.

The garbage collection rules are defined in Figures 12 through 15. These in turn deal with the garbage collection of types, values, environments, and stacks. The garbage collection of types is defined by a series of rules of the form $(Hf,\ Ht_1,\ PF_1,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ p_2)$. These rules perform a recursive copy of the type referenced by p_1 (in Hf) into Ht. The final state contains the copied type in Ht_2, a new pointer to the type p_2, and entries in PF_2 for the copied type. There are separate rules for each of the $\mathcal{M}\Lambda$ types. Rule 9 ensures that types are only copied once.

$$\frac{p_1 \in \mathrm{Dom}\ PF}{(Hf,\ Ht,\ PF,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht,\ PF,\ LF,\ PF(p_1))} \tag{9}$$

$$\frac{p_1 \notin \mathrm{Dom}\ PF \qquad Hf(p_1) = tn}{(Hf,\ Ht,\ PF,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht[p_2 \mapsto tn],\ PF[p_1 \mapsto p_2],\ LF,\ p_2)} \tag{10}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\ PF_1 \qquad Hf(p_1) = tn(p_2) \\ (Hf,\ Ht_1,\ PF_1,\ LF,\ p_2) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ p_3) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2[p_4 \mapsto tn(p_3)],\ PF_2[p_1 \mapsto p_4],\ LF,\ p_4) \end{array}} \tag{11}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\ PF_1 \qquad Hf(p_1) = \overline{p_2}^{\,k} \\ (Hf,\ Ht_1,\ PF_1,\ LF,\ p_2^1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ p_3^1) \cdots \\ \quad (Hf,\ Ht_k,\ PF_k,\ LF,\ p_2^k) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF,\ p_3^k) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_{k+1}[p_4 \mapsto (p_3^1,\ \ldots,\ p_3^k)],\ PF_{k+1}[p_1 \mapsto p_4],\ LF,\ p_4) \end{array}} \tag{12}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\ PF_1 \qquad Hf(p_1) = p_2 \rightarrow p_3 \\ (Hf,\ Ht_1,\ PF_1,\ LF,\ p_2) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ p_4) \\ (Hf,\ Ht_2,\ PF_2,\ LF,\ p_3) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF,\ p_5) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_3[p_6 \mapsto p_4 \rightarrow p_5],\ PF_3[p_1 \mapsto p_6],\ LF,\ p_6) \end{array}} \tag{13}$$

$$\frac{p_1 \notin \mathrm{Dom}\ PF \qquad Hf(p_1) = tv}{(Hf,\ Ht,\ PF,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht[p_2 \mapsto tv],\ PF[p_1 \mapsto p_2],\ LF,\ p_2)} \tag{14}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\ PF_1 \qquad Hf(p_1) = \forall\, \overline{p_2}^{\,k}.\ p_3 \\ (Hf,\ Ht_1,\ PF_1,\ LF,\ p_2^1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ p_4^1) \cdots \\ \quad (Hf,\ Ht_k,\ PF_k,\ LF,\ p_2^k) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF,\ p_4^k) \\ (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF,\ p_3) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+2},\ PF_{k+2},\ LF,\ p_5) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_{k+2}[p_6 \mapsto \forall\, \overline{p_4}^{\,k}.\ p_5],\ PF_{k+2}[p_1 \mapsto p_6],\ LF,\ p_6) \end{array}} \tag{15}$$

Figure 12: Type Heap Rules.

$$\frac{l_1 \in \mathrm{Dom}\ LF}{(Hf,\ Ht,\ PF,\ LF,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht,\ PF,\ LF,\ (LF(l_1),\ p_1))} \tag{16}$$

$$\frac{l_1 \notin \mathrm{Dom}\ LF \qquad Hf(p_1) = tn}{\begin{array}{l}(Hf,\ Ht,\ PF,\ LF,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht[l_2 \mapsto Hf(l_1)],\ PF,\ LF[l_1 \mapsto l_2],\ (l_2,\ p_1))\end{array}} \tag{17}$$

$$\frac{\begin{array}{l}l_1 \notin \mathrm{Dom}\ LF_1 \qquad Hf(p_1) = tn(p_2) \qquad Hf(l_1) = con(l_2) \\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_2,\ p_2)) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ (l_3,\ p_2))\end{array}}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2[l_4 \mapsto con(l_3)],\ PF_2,\ LF_2[l_1 \mapsto l_4],\ (l_4,\ p_1))\end{array}} \tag{18}$$

$$\frac{\begin{array}{l}l_1 \notin \mathrm{Dom}\ LF_1 \qquad Hf(p_1) = \overline{p_2}^{\,k} \qquad Hf(l_1) = \overline{l_2}^{\,k} \\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_2^1,\ p_2^1)) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ (l_3^1,\ p_2^1)) \cdots \\ \quad (Hf,\ Ht_k,\ PF_k,\ LF_k,\ (l_2^k,\ p_2^k)) \Rightarrow_{\mathrm{gc}} \\ \qquad (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF_{k+1},\ (l_3^k,\ p_2^k))\end{array}}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_{k+1}[l_4 \mapsto (l_3^1,\ \ldots,\ l_3^k)],\ PF_{k+1},\ LF_{k+1}[l_1 \mapsto l_4],\ (l_4,\ p_1))\end{array}} \tag{19}$$

$$\frac{\begin{array}{l}l_1 \notin \mathrm{Dom}\ LF_1 \qquad Hf(p_1) = p_2 \to p_3 \qquad Hf(l_1) = \langle\!\langle E_1,\ \overline{lv}^{\,k},\ \Lambda\rangle\!\rangle \\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ E_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ E_2)\end{array}}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2[l_2 \mapsto \langle\!\langle E_2,\ \overline{lv}^{\,k},\ \Lambda\rangle\!\rangle],\ PF_2,\ LF_2[l_1 \mapsto l_2],\ (l_2,\ p_1))\end{array}} \tag{20}$$

$$\frac{l_1 \notin \mathrm{Dom}\ LF_1 \qquad Hf(p_1) = p_2 \to p_3 \qquad Hf(l_1) = \Omega}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2[l_2 \mapsto \Omega],\ PF_2,\ LF_2[l_1 \mapsto l_2],\ (l_2,\ p_1))\end{array}} \tag{21}$$

Figure 13: Value Heap Rules.

The garbage collection of values is defined in Figure 13 by rules of the form $(Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ (l_2,\ p_2))$. These rules perform a recursive copy of the value referenced by l_1 into Ht_1. This uses the type referenced by p_1 to determine the shape. The result is a new location l_2 which references the copied value in Ht_2. There is a separate rule for each of the $\mathcal{M}\Lambda$ values. In the case where the value is a closure (Rule 20), the rules in Figure 14 are used to copy the closure environment.

The collection rules for environments are given in Figure 14. These rules have the form $(Hf,\ Ht_1,\ PF_1,\ LF_1,\ E_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ E_2)$. The environment E_1 is decomposed into a constructor environment CE_1, and a value environment VE_1. A new constructor environment CE_2 is built by copying all of the types referenced in CE_1. Similarly, a new value environment VE_2 is built by copying all of the values and types referenced in VE_1. Finally, a new environment E_2 is built from CE_2, and VE_2.

$$\frac{\begin{array}{l} E_1 = (TE,\ CE_1,\ VE_1) \\ CE_1 = \{con^1 \ \mapsto\ p_1^1,\ \ldots,\ con^k \ \mapsto\ p_1^k\} \\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ p_1^1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_1,\ p_2^1) \ \cdots \\ \quad (Hf,\ Ht_k,\ PF_k,\ LF_1,\ p_1^k) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF_1,\ p_2^k) \\ CE_2 = \{con^1 \ \mapsto\ p_2^1,\ \ldots,\ con^k \ \mapsto\ p_2^k\} \\ VE_1 = \{lv^1 \ \mapsto\ (l_1^1,\ p_1^1),\ \ldots,\ lv^l \ \mapsto\ (l_1^l,\ p_1^l)\} \\ (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF_1,\ p_1^1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+2},\ PF_{k+2},\ LF_1,\ p_2^1) \ \cdots \\ \quad (Hf,\ Ht_{k+l},\ PF_{k+l},\ LF_1,\ p_1^l) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+l+1},\ PF_{k+l+1},\ LF_1,\ p_2^l) \\ (Hf,\ Ht_{k+l+1},\ PF_{k+l+1},\ LF_1,\ (l_1^1,\ p_1^1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_{k+l+2},\ PF_{k+l+2},\ LF_2,\ (l_2^1,\ p_1^1)) \ \cdots \\ \qquad (Hf,\ Ht_{k+2l},\ PF_{k+2l},\ LF_l,\ (l_1^l,\ p_1^l)) \Rightarrow_{\mathrm{gc}} \\ \qquad\quad (Hf,\ Ht_{k+2l+1},\ PF_{k+2l+1},\ LF_{l+1},\ (l_2^l,\ p_1^l)) \\ VE_2 = \{lv^1 \ \mapsto\ (l_2^1,\ p_2^1),\ \ldots,\ lv^l \ \mapsto\ (l_2^l,\ p_2^l)\} \\ E_2 = (TE,\ CE_2,\ VE_2) \end{array}}{(Hf,\ Ht_1,\ PF_1,\ LF_1,\ E_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+2l+1},\ PF_{k+2l+1},\ LF_{l+1},\ E_2)} \tag{22}$$

Figure 14: Environment Rule.

The garbage collection rules in Figure 15 are used to copy a stack recursively. The item at the top of the stack is either a type, a value, or an environment. This will cause the invocation of the corresponding rule from one of the previous three figures. The four rules for collecting stacks all have the form $(Hf,\ Ht_1,\ PF_1,\ LF_1,\ S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ S_2)$.

$$\frac{}{(Hf,\ Ht,\ PF,\ LF,\ ()) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht,\ PF,\ LF,\ ())} \tag{23}$$

$$\frac{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_1,\ p_2) \\ (Hf,\ Ht_2,\ PF_2,\ LF_1,\ S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_2,\ S_2) \end{array}}{(Hf,\ Ht_1,\ PF_1,\ LF_1,\ p_1{\cdot}S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_2,\ p_2{\cdot}S_2)} \tag{24}$$

$$\frac{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_1,\ p_2) \\ (Hf,\ Ht_2,\ PF_2,\ LF_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_2,\ (l_2,\ p_1)) \\ (Hf,\ Ht_3,\ PF_3,\ LF_2,\ S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_4,\ PF_4,\ LF_3,\ S_2) \end{array}}{(Hf,\ Ht_1,\ PF_1,\ LF_1,\ (l_1,\ p_1){\cdot}S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_4,\ PF_4,\ LF_3,\ (l_2,\ p_2){\cdot}S_2)} \tag{25}$$

$$\frac{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ E_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ E_2) \\ (Hf,\ Ht_2,\ PF_2,\ LF_2,\ S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_3,\ S_2) \end{array}}{(Hf,\ Ht_1,\ PF_1,\ LF_1,\ E_1{\cdot}S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_3,\ E_2{\cdot}S_2)} \tag{26}$$

Figure 15: Stack Rules.

4 Distributed Memory Management

The LEMMA memory interface [2] identifies two main services that are required for distributed memory management: sharing of distributed data, and distributed garbage collection. As far as possible, the LEMMA interface has been designed to be independent of the details of the language implemented on it, and also as independent as possible of the memory and communication system on which it is implemented. The interface definition is simply a function-level specification, the actual implementation is left open. Different implementations of the interface have been created for parallel computers [11] and local-area networks of workstations [12]. A wide-area network implementation is also in progress. In each case, the assumptions regarding the speed and reliability of the underlying network require a different solution. Here the focus is on providing an abstract machine definition suitable for the LEMMA interface on a reliable local-area network. However, the resulting definition can easily be extended to cover a range of possibilities.

The LEMMA interface is based on the Distributed Shared Memory (DSM) model [13]. In this model, the memory of the distributed system is treated as a single globally-addressable object. LEMMA statically partitions the global address space into contiguous semi-spaces, where each semi-space is managed by a different machine. Each machine is responsible for allocation and garbage-collection within its own semi-space.

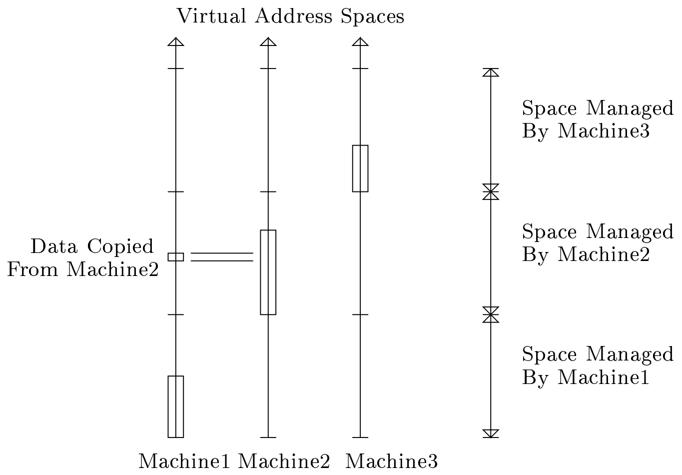

Figure 16: Distributed Memory Access.

In a parallel implementation, where communication costs are low, data accesses are simply performed in-place in the global address space. However, in a workstation implementation, the overheads associated with distributed memory access necessitate a form of caching mechanism. The solution adopted by the workstation implementation of LEMMA uses the fact that a typical workstation has a very large virtual address space, but only uses a fraction of it for its real memory. Thus, the virtual address space on each machine is used to represent the entire distributed address space. When a machine wishes to access some external data, it simply addresses its own virtual address space at the required location. The page-faulting mechanism of the operating system is extended to fetch the data from the remote machine, as illustrated in Figure 16. Further accesses to the data will simply use the local copy. This operation can be implemented very efficiently on a typical UNIX workstation.

It is clear that some form of coherency protocol is also required. The original data may be updated, invalidating any copies on other machines. A number of schemes for ensuring coherency are detailed in [14]. One significant property of typical Standard ML programs is that most of the data values are immutable. The only values on which updates are permitted are references. Thus, LEMMA distinguishes between mutables and immutables and only performs coherency checks on the mutables. For brevity, only the caching of immutables is formalised in this report. Mutables are simply accessed in-place remotely. A formalisation of the caching and coherency mechanism for the mutables is left as further work.

The provision of efficient algorithms for distributed garbage collection continues to be a very active area of garbage collection research. Techniques vary from simple schemes (e.g. where one process garbage collections while another executes), up to complex multi-generational schemes. A survey of distributed garbage collection techniques is presented in [15]. The technique presented here is not the most efficient, but extends naturally from the sequential case described in the previous section. A number of techniques for improving the efficiency of the algorithm are outlined in [12].

Each semi-space of the global address space managed by each machine is further divided in two to constitute the *from* and *to* spaces. The collective *from* and *to* spaces are the concatenation of these local spaces. Garbage collection begins with a global synchronisation. All of the machines perform garbage collection in parallel, but asynchronously. The task of each machine is to ensure that all of the objects in the collective *from* space that are reachable from its own roots have been copied into the collective *to* space. Once all of the machines have finished, the global garbage collection is complete and each machine resumes executing in its local *to* space.

The garbage collection algorithm executed by each machine is the same as the sequential variant with the following addition:

- When machine A encounters a pointer to an object managed by machine B, it sends a message containing the pointer to machine B.
- If the object has already been copied, machine B returns the updated pointer, so machine A can update the object that it was scanning.
- If the object has not been copied, machine B proceeds with the copy and returns the updated pointer to machine A. During the copy, machine B may encounter further remote pointers, in which case the above steps are repeated.

The protocol has the property that an object is always copied by the machine that first created it, even if that machine no longer has a reference to it. It must also be noted that the garbage collection invalidates all of the cached copies of objects on other machines. Optimisations which permit object migration, and provide cache contents preservation are described in [12] although they will not be formalised here.

4.1 Distributed Abstract Machine

The provision of an abstract machine definition for the complete $\mathcal{M}\Lambda$ language introduces a number of interesting problems. The language includes primitives for providing communication and concurrency. However, it is far from obvious how these should be expressed in the abstract machine semantics. An examination of the literature on concurrency in structured operational semantics reveals that a 'single-step' (atomic action) approach is typically used to express the computation. Concurrency is then introduced by interleaving the atomic actions of the processes. Unfortunately, this approach is directly in contrast with the abstract machine style, where the intermediate steps are hidden and only the final state of the computation is of interest.

At first glance, it appears that the semantics must be rewritten in the single-step style before the concurrent extensions can be expressed. This is undesirable for a number of reasons. The abstract machine model provides an ideal setting for expressing memory management behaviour in a manner that approximates an actual implementation. Furthermore, the single-step approach requires a large number of rules for even the simplest of operations. A further problem can occur when some of the intermediate states do not correspond directly to programs in the language. In such cases, the language may need to be extended with additional constructs before the semantics can be defined. Techniques exist for automatically performing this transformation in a number of cases [16]. However, the transformed semantics is rather messy in comparison to the original, and this makes reasoning about the extended language more tedious.

Fortunately, there is an alternative technique inspired by the relational-style semantics of concurrency presented in [17] and [18]. Consider a transition of the form: $(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ RS_2)$. This transition expresses the evaluation an expression Λ with machine state $(H_1,\ E_1,\ ES_1,\ RS_1)$ resulting in the final machine state $(H_2,\ E_2,\ ES_2,\ RS_2)$. In the full language, this treatment is no longer acceptable as the evaluation of Λ may require some external interaction before it can yield the result. This leads to a definition of the form $(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \stackrel{t}{\Rightarrow} (H_2,\ E_2,\ ES_2,\ RS_2)$, where t is a trace which records the external interactions required to produce the final result.

$$\text{Trace} \quad t \quad ::= \quad \epsilon \quad | \quad t_1\ ;\ t_2 \quad | \quad t_1 \parallel t_2 \quad | \quad cn\ !\ m \quad | \quad cn\ ?\ m$$

Figure 17: Computation and Communication Traces.

For the $\mathcal{M}\Lambda$ language, the definition of traces in Figure 17 is sufficient. A trace may be empty ϵ if no external interactions occur. The sequential evaluation of two expressions leads to a trace of the form $t_1\ ;\ t_2$. Similarly, two expressions evaluated in parallel produce a trace of the form $t_1 \parallel t_2$. A message m sent over a channel cn yields a trace of the form $cn\ !\ m$, and a message received on the channel gives the trace $cn\ ?\ m$. Figure 18 contains the reduction rules for traces. It should always be the case that the traces reduce to ϵ for a complete program, or an error will have occurred.

$$\epsilon \; ; \; t \quad = \quad t \tag{27}$$

$$\epsilon \parallel t \quad = \quad t \tag{28}$$

$$t_1 \parallel t_2 \quad = \quad t_2 \parallel t_1 \tag{29}$$

$$cn \; ! \; m \parallel cn \; ? \; m \quad = \quad \epsilon \tag{30}$$

$$cn \; ! \; m \; ; \; t_1 \parallel cn \; ? \; m \; ; \; t_2 \quad = \quad t_1 \parallel t_2 \tag{31}$$

Figure 18: Trace Reduction Rules.

The syntax of the distributed abstract machine, hereafter referred to as the $\mathcal{D}M\Lambda$ abstract machine, is given in Figure 20. In order to remain consistent with the LEMMA interface definition, the machine contains two levels of abstraction: *processes* and *threads*. The sequential abstract machine defined in Section 3 effectively corresponds to a single thread executing on a machine containing a single process.

There are a fixed number k of concurrently executing processes Π each containing a local heap H, a cache C, and a multi-set of threads $\overline{\overline{\mathcal{T}}}$. It is assumed that these processes are physically distributed, for example, across a network of k machines. The memory of the machine is modelled using distributed shared memory. This means that any process can directly access the data contained on the heap of any other process, e.g. $H^i(p) = ty$, where $0 < i \leq k$.

The set of threads associated with each process also execute concurrently. However, unlike processes, the multi-set of threads may dynamically grow or shrink. The multi-set may also be empty. Each thread contains a local environment E, an exception stack ES, and a result stack RS. The threads share the heap of the parent process.

The execution of processes and threads is defined in Figure 19. All of the processes execute concurrently as defined in Rule 32. Note the parallel composition of the traces generated by each process. Concurrent execution of threads is defined inductively in Rule 33. The base case, an empty multi-set of threads, is defined by Rule 34.

$$\frac{\Pi_1^1 \stackrel{t_1}{\Longrightarrow} \Pi_2^1 \quad \cdots \quad \Pi_1^k \stackrel{t_k}{\Longrightarrow} \Pi_2^k}{(\Pi_1^1, \; \ldots, \; \Pi_1^k) \stackrel{t_1 \parallel \cdots \parallel t_k}{\Longrightarrow} (\Pi_2^1, \; \ldots, \; \Pi_2^k)} \tag{32}$$

$$\frac{\mathcal{T}_1 \stackrel{t_1}{\Longrightarrow} \mathcal{T}_3 \qquad \overline{\overline{\mathcal{T}_2}} \stackrel{t_2}{\Longrightarrow} \overline{\overline{\mathcal{T}_4}}}{\mathcal{T}_1 \uplus \overline{\overline{\mathcal{T}_2}} \stackrel{t_1 \parallel t_2}{\Longrightarrow} \mathcal{T}_3 \uplus \overline{\overline{\mathcal{T}_4}}} \tag{33}$$

$$\frac{}{\emptyset \stackrel{\epsilon}{\Longrightarrow} \emptyset} \tag{34}$$

Figure 19: Concurrent Execution of Processes and Threads.

Machine State	M	::=	$\overline{\Pi}^k$
Process	Π	::=	$(H,\ C,\ \overline{\overline{\mathcal{T}}})$
Thread	$\mathcal{T}$	::=	$(E,\ ES,\ RS)$
Trace	t	::=	$\epsilon \mid t_1\ ;\ t_2 \mid t_1 \parallel t_2 \mid cn\ !\ m \mid cn\ ?\ m$
Message	m	::=	$(p) \mid (l,\ p) \mid (E)$
Heap	H	::=	$(TH,\ VH)$
Cache	C	::=	$(TC,\ VC)$
Type Heap	TH	::=	$p \overset{map}{\mapsto} ty$
Type Cache	TC	::=	$p \overset{map}{\mapsto} ty$
Heap Types	ty	::=	$tn \mid tn(p) \mid$ $\overline{p}^k \mid p_1 \to p_2 \mid$ $tv \mid \forall\, \overline{p_1}^k\ .\ p_2$
Value Heap	VH	::=	$l \overset{map}{\mapsto} val$
Value Cache	VC	::=	$l \overset{map}{\mapsto} val$
Heap Values	val	::=	$scon \mid cn \mid con \mid con(l) \mid$ $\overline{l}^k \mid \langle\!\langle E,\ \overline{lv}^k,\ \Lambda \rangle\!\rangle \mid \Omega$
Environment	E	::=	$(TE,\ NE,\ CE,\ VE)$
Type Env.	TE	::=	$\overline{\overline{tn}}$
Channel Env.	NE	::=	$\overline{\overline{cn}}$
Constructor Env.	CE	::=	$con \overset{map}{\mapsto} p$
Variable Env.	VE	::=	$lv \overset{map}{\mapsto} (l,\ p)$
Exception Stack	ES	::=	$() \mid (l,\ p)\cdot ES$
Result Stack	RS	::=	$() \mid p\cdot RS \mid (l,\ p)\cdot RS \mid E\cdot RS$

Figure 20: $\mathcal{D}M\Lambda$ Abstract Machine Syntax.

The memory of each process is represented by the heap H and the cache C. With reference to Figure 16, the area of the virtual address space which corresponds to the area managed by the process is represented by the heap. The remainder of the virtual address space is represented by the cache. The allocation of heap values and types is performed in two stages in the $\mathcal{D}M\Lambda$ machine. $H \uparrow l$ returns the next available heap location in H, and $H[l \mapsto val]$ allocates val on the heap and binds it to the location l. Similarly, $H \uparrow p$ returns the next available type-pointer, and $H[p \mapsto ty]$ allocates ty on the heap bound to the pointer p. Thus, heap locations and type-pointers may effectively be reserved before being used. Data is allocated on the cache in a single step, e.g. $C[l \mapsto val]$ allocates val in the cache at location l. Cache locations and pointers do not need to be reserved as they correspond directly to locations and pointers in the heaps (it is assumed that heap locations and pointers belonging to different heaps are distinct).

4.2 Data Caching

The caching mechanisms of the LEMMA interface are formalised by the rules in Figures 21 and 22. In order for these mechanisms to be effective, all memory accesses in the $\mathcal{DM}\Lambda$ machine are done via a *fetch* function. The arguments to this function are either a type-pointer p or a pair $(l,\ p)$.

The rules are defined via transitions between machine states of the form $(\Pi^1,\ \ldots,\ \Pi^k)$. Each of the rules is defined for a single *fetch* operation within a single thread n on a process Π^i, where $0 < i \leq k$. The rules utilise the functions $num(p)$ and $num(l)$ which return the number of the process whose heap contains the type-pointer p or heap-location l respectively. The $mutable(p)$ predicate determines if a pointer p references a mutable type. In an implementation, the heap is typically divided into mutable and immutable areas. Thus, this predicate may be implemented by simply examining the address of the pointer.

The $fetch(p)$ function returns the type referenced by the type-pointer p. This function is defined in Figure 21 by four rules corresponding to the following cases. The type may be local, i.e. contained within the heap of the parent process, in which case it is accessed directly (Rule 35). The type may be non-local and mutable in which case, it is accessed remotely in-place (Rule 36). The type may be remote, but a local copy may exist in the cache, in which case the cached version is accessed (Rule 37). Finally, if the type is remote and immutable, and a copy is not present in the cache, then a local copy is made (Rule 38).

$$\frac{num(p) = i}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k) \stackrel{\epsilon}{\Longrightarrow} \\ \quad (\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ H^i(p) \cdot RS^n) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k)\end{array}} \tag{35}$$

$$\frac{num(p) = j \qquad j \neq i \qquad mutable(p)}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \quad (H^j,\ C^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k) \stackrel{\epsilon}{\Longrightarrow} \\ \qquad (\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ H^j(p) \cdot RS^n) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \qquad\quad (H^j,\ C^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k)\end{array}} \tag{36}$$

$$\frac{num(p) = j \qquad j \neq i \qquad \neg mutable(p) \qquad p \in \mathrm{Dom}\ C^i}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k) \stackrel{\epsilon}{\Longrightarrow} \\ \quad (\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ C^i(p) \cdot RS^n) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k)\end{array}} \tag{37}$$

$$\frac{num(p) = j \qquad j \neq i \qquad \neg mutable(p) \qquad p \notin \mathrm{Dom}\ C^i \qquad H^j(p) = ty}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \quad (H^j,\ C^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k) \stackrel{\epsilon}{\Longrightarrow} \\ \qquad (\Pi^1,\ \ldots,\ (H^i,\ C^i[p\ \mapsto\ ty],\ (E^n,\ ES^n,\ ty \cdot RS^n) \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \qquad\quad (H^j,\ C^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k)\end{array}} \tag{38}$$

Figure 21: Type Heap Caching.

The $fetch(l,\ p)$ function is described in Figure 22. The function returns a pair $(val,\ ty)$ containing the value and the type corresponding to the location and pointer of the arguments. Once again, this operation is defined by four rules, corresponding to the four cases described previously: local, mutable, cached, and remote. The rules are slightly more complex as the type must first be obtained via a call to $fetch(p)$ before the value is retrieved.

$$\frac{\begin{array}{l} num(l) = i \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E^n,\ ES^n,\ ty{\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(l,\ p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E^n,\ ES^n,\ (H_2^i(l),\ ty){\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{39}$$

$$\frac{\begin{array}{l} num(l) = j \qquad j \neq i \qquad mutable(p) \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E^n,\ ES^n,\ ty{\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \quad\quad (H_1^j,\ C_1^j,\ \overline{\overline{\overline{\mathcal{T}_1^j}}}),\ \ldots,\ \Pi_2^k) \\ H_1^j(l) = val \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(l,\ p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E^n,\ ES^n,\ (val,\ ty){\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \quad\quad (H_1^j,\ C_1^j,\ \overline{\overline{\overline{\mathcal{T}_1^j}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{40}$$

$$\frac{\begin{array}{l} num(l) = j \qquad j \neq i \qquad \neg mutable(p) \qquad l \in \mathrm{Dom}\ C_1^i \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E^n,\ ES^n,\ ty{\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \\ C_2^i(l) = val \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(l,\ p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E^n,\ ES^n,\ (val,\ ty){\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{41}$$

$$\frac{\begin{array}{l} num(l) = j \qquad j \neq i \qquad \neg mutable(p) \qquad l \notin \mathrm{Dom}\ C_1^i \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E^n,\ ES^n,\ ty{\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \quad\quad (H_1^j,\ C_1^j,\ \overline{\overline{\overline{\mathcal{T}_1^j}}}),\ \ldots,\ \Pi_2^k) \\ H_1^j(l) = val \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ ES^n,\ RS^n,\ fetch(l,\ p)) \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i[l\ \mapsto\ val],\ (E^n,\ ES^n,\ (val,\ ty){\cdot}RS^n) \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \quad\quad (H_1^j,\ C_1^j,\ \overline{\overline{\overline{\mathcal{T}_1^j}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{42}$$

Figure 22: Value Heap Caching.

4.3 Concurrency and Communication

The dynamic semantics of the complete $\mathcal{M}\Lambda$ language are formalised by transitions between states of the distributed abstract machine. As with the caching rules, the transitions are defined for a single thread n executing on a process Π^i, where $0 < i \leq k$.

The initial state of the $\mathcal{D}M\Lambda$ abstract machine is illustrated in Figure 23. Evaluation begins with a single process Π^1, containing the initial heap H^1, and a single thread $(E^1,\ (),\ (),\ P)$, containing the initial environment E^1 and the program program P. The remaining processes $\Pi^2\ \cdots\ \Pi^k$ remain idle, with an empty heap and multi-set of threads, until an rfork is performed by the program.

$$
\begin{array}{lcl}
M & = & (\Pi^1,\ (\emptyset,\ \emptyset,\ \emptyset)^2,\ \ldots,\ (\emptyset,\ \emptyset,\ \emptyset)^k) \\
\Pi^1 & = & (H^1,\ \emptyset,\ \{(E^1,\ (),\ (),\ P)\}) \\
 & & \\
H^1 & = & (\mathit{TH},\ \emptyset) \\
\mathit{TH} & = & \{p_1\ \mapsto\ \mathrm{t\_unit} \rightarrow \mathrm{t\_bool}, \\
 & & \ p_2\ \mapsto\ tv^1,\quad p_3\ \mapsto\ \forall\ (p_2)\ .\ \mathrm{t\_unit} \rightarrow \mathrm{t\_list}(p_2), \\
 & & \ p_4\ \mapsto\ tv^2,\quad p_5\ \mapsto\ \forall\ (p_4)\ .\ (p_4,\ \mathrm{t\_list}(p_4)) \rightarrow \mathrm{t\_list}(p_4), \\
 & & \ p_6\ \mapsto\ tv^3,\quad p_7\ \mapsto\ \forall\ (p_6)\ .\ p_6 \rightarrow \mathrm{t\_ref}(p_6), \\
 & & \ p_8\ \mapsto\ \mathrm{t\_unit} \rightarrow \mathrm{t\_exn}\} \\
 & & \\
E^1 & = & (\mathit{TE},\ \emptyset,\ \mathit{CE},\ \emptyset) \\
\mathit{TE} & = & \{\mathrm{t\_unit},\ \mathrm{t\_int},\ \mathrm{t\_word},\ \mathrm{t\_real},\ \mathrm{t\_char},\ \mathrm{t\_string},\ \mathrm{t\_bool}, \\
 & & \ \mathrm{t\_list},\ \mathrm{t\_ref},\ \mathrm{t\_exn},\ \mathrm{t\_chan}\} \\
\mathit{CE} & = & \{\mathrm{c\_true}\ \mapsto\ p_1,\ \mathrm{c\_false}\ \mapsto\ p_1, \\
 & & \ \mathrm{c\_nil}\ \mapsto\ p_3,\ \mathrm{c\_cons}\ \mapsto\ p_5,\ \mathrm{c\_ref}\ \mapsto\ p_7, \\
 & & \ \mathrm{e\_match}\ \mapsto\ p_8,\ \mathrm{e\_bind}\ \mapsto\ p_8\}
\end{array}
$$

Figure 23: Initial Machine State.

There is a relatively straightforward conversion from the transitions of the $\mathcal{M}\Lambda$ machine to the $\mathcal{D}M\Lambda$ machine. This conversion is performed via the templates illustrated in Figure 24.

The first template (43) is used to convert an $\mathcal{M}\Lambda$ program into a $\mathcal{D}M\Lambda$ program. A $\mathcal{M}\Lambda$ heap H becomes a $\mathcal{D}M\Lambda$ heap H^i. Similarly, am $\mathcal{M}\Lambda$ environment E becomes a $\mathcal{D}M\Lambda$ environment E^n, etc. The templates for datatypes and exceptions are essentially identical and are not given here. The template for expressions (44) is also very similar.

Owing to the shared-memory model of the $\mathcal{D}M\Lambda$ machine, $\mathcal{M}\Lambda$ memory accesses must be converted into calls to the $fetch$ function. This is achieved simply using the templates (45) and (46). There is one slight complication in that a garbage collection can now occur during a $fetch$ operation. Therefore, some additional saving and restoring on the result stack is required to ensure that all pointers will be collected. This is illustrated in the example below.

With a little work, these templates can be used to convert all of the $\mathcal{M}\Lambda$ rules in Appendix A into their $\mathcal{D}M\Lambda$ counterparts. For brevity, this is left as an exercise for the reader. The conversion simply involves renumbering the heaps, stacks, etc. and ensuring that type-pointers and value-locations are saved and restored on the result stack around instances of the $fetch$ operation.

$$\begin{array}{lll}\mathcal{M}\Lambda: & (H_1,\ E_1,\ ES_1,\ RS_1,\ P) \Rightarrow (H_2,\ E_2,\ ES_2,\ RS_2) & \\ \mathcal{DM}\Lambda: & (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ P) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} & (43) \\ & \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ RS_2^n) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) & \end{array}$$

$$\begin{array}{lll}\mathcal{M}\Lambda: & (H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l,\ p)\cdot RS_2) & \\ \mathcal{DM}\Lambda: & (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ \Lambda) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} & (44) \\ & \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ (l,\ p)\cdot RS_2^n) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) & \end{array}$$

$$\begin{array}{lll}\mathcal{M}\Lambda: & H_1(p) = ty & \\ \mathcal{DM}\Lambda: & (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ fetch(p)) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Longrightarrow} & (45) \\ & \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ (ty)\cdot RS_2^n) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) & \end{array}$$

$$\begin{array}{lll}\mathcal{M}\Lambda: & H_1(p) = ty \qquad H_1(l) = val & \\ \mathcal{DM}\Lambda: & (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ fetch(l,\ p)) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) & (46) \\ & \quad \overset{t}{\Longrightarrow} (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ (val,\ ty)\cdot RS_2^n) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) & \end{array}$$

Figure 24: Conversion Templates from $\mathcal{M}\Lambda$ to $\mathcal{DM}\Lambda$.

As an example, the $\mathcal{M}\Lambda$ **app** expression (Rule 101) is converted into the $\mathcal{DM}\Lambda$ abstract machine according to Figure 25. Note the sequential composition of the evaluation traces for the subexpressions: $t_1\ ;\ t_2\ ;\ t_3\ ;\ t_4$. This must be performed for all of the rules in the sequential sub-language.

$$\frac{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ \Lambda_1) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \overset{t_1}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ RS_2^n) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \\ (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ RS_2^n,\ \Lambda_2) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \overset{t_2}{\Longrightarrow} \\ \quad (\Pi_3^1,\ \ldots,\ (H_3^i,\ C_3^i,\ (E_3^n,\ ES_3^n,\ (l_2,\ p_2)\cdot(l_1,\ p_1)\cdot RS_3^n) \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k) \\ (\Pi_3^1,\ \ldots,\ (H_3^i,\ C_3^i,\ (E_3^n,\ ES_3^n,\ (l_2,\ p_2)\cdot RS_3^n,\ fetch(l_1,\ p_1)) \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k) \\ \quad \overset{t_3}{\Longrightarrow} (\Pi_4^1,\ \ldots,\ (H_4^i,\ C_4^i,\ (E_4^n,\ ES_4^n, \\ \qquad (\langle\!\langle E_c,\ lv,\ \Lambda_c\rangle\!\rangle,\ ty)\cdot(l_3,\ p_3)\cdot RS_4^n) \uplus \overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k) \\ (\Pi_4^1,\ \ldots,\ (H_4^i,\ C_4^i,\ (E_c[lv\ \mapsto\ (l_3,\ p_3)],\ ES_4^n,\ E_4^n\cdot RS_4^n,\ \Lambda_c) \uplus \overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k) \\ \quad \overset{t_4}{\Longrightarrow} (\Pi_5^1,\ \ldots,\ (H_5^i,\ C_5^i,\ (E_5^n,\ ES_5^n,\ (l_4,\ p_4)\cdot E_6\cdot RS_4^n) \uplus \overline{\overline{\mathcal{T}_5^i}}),\ \ldots,\ \Pi_5^k)\end{array}}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ \mathbf{app}\ (\Lambda_1,\ \Lambda_2)) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \\ \quad \overset{t_1\ ;\ t_2\ ;\ t_3\ ;\ t_4}{\Longrightarrow} (\Pi_5^1,\ \ldots,\ (H_5^i,\ C_5^i,\ (E_6,\ ES_5^n,\ (l_4,\ p_4)\cdot RS_5^n) \uplus \overline{\overline{\mathcal{T}_5^i}}),\ \ldots,\ \Pi_5^k)\end{array}} \qquad (47)$$

Figure 25: Function Application in $\mathcal{DM}\Lambda$.

$$\frac{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ \Lambda) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \stackrel{t_1}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ (l_1,\ p_1)\cdot RS_2^n) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \\ (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ RS_2^n,\ fetch(l_1,\ p_1)) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \stackrel{t_2}{\Longrightarrow} \\ \quad (\Pi_3^1,\ \ldots,\ (H_3^i,\ C_3^i,\ (E_3^n,\ ES_3^n,\ (\langle\!\langle E_c,\ lv,\ \Lambda_c\rangle\!\rangle,\ ty)\cdot RS_3^n) \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k)\end{array}}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ \mathbf{fork}(\Lambda)) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \stackrel{t_1\ ;\ t_2}{\Longrightarrow} \\ \quad (\Pi_3^1,\ \ldots,\ (H_3^i,\ C_3^i,\ (E_3^n,\ ES_3^n,\ (l_{unit},\ p_{unit})\cdot RS_3^n) \uplus \\ \quad\quad (E_c,\ \emptyset,\ \emptyset,\ \Lambda_c) \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k)\end{array}} \tag{48}$$

$$\frac{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ \Lambda_1) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \stackrel{t_1}{\Longrightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ (l_1,\ p_1)\cdot RS_2^n) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \\ (\Pi_2^1,\ \ldots,\ (H_2^i,\ C_2^i,\ (E_2^n,\ ES_2^n,\ RS_2^n,\ fetch(l_1,\ p_1)) \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \stackrel{t_2}{\Longrightarrow} \\ \quad (\Pi_3^1,\ \ldots,\ (H_3^i,\ C_3^i,\ (E_3^n,\ ES_3^n,\ RS_3^n) \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k) \\ (\Pi_3^1,\ \ldots,\ (H_3^i,\ C_3^i,\ (E_3^n,\ ES_3^n,\ RS_3^n,\ \Lambda_2) \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k) \stackrel{t_3}{\Longrightarrow} \\ \quad (\Pi_4^1,\ \ldots,\ (H_4^i,\ C_4^i,\ (E_4^n,\ ES_4^n,\ (l_2,\ p_2)\cdot(j,\ ty_1)\cdot RS_4^n) \uplus \overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k) \\ (\Pi_4^1,\ \ldots,\ (H_4^i,\ C_4^i,\ (E_4^n,\ ES_4^n,\ RS_4^n,\ fetch(l_2,\ p_2)) \uplus \overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k) \stackrel{t_4}{\Longrightarrow} \\ \quad (\Pi_5^1,\ \ldots,\ (H_5^i,\ C_5^i,\ (E_5^n,\ ES_5^n,\ (\langle\!\langle E_c,\ lv,\ \Lambda_c\rangle\!\rangle,\ ty_2)\cdot RS_5^n) \uplus \overline{\overline{\mathcal{T}_5^i}}),\ \ldots \\ \quad\quad (H^j,\ C^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi_5^k)\end{array}}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E_1^n,\ ES_1^n,\ RS_1^n,\ \mathbf{rfork}(\Lambda_1,\ \Lambda_2)) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \\ \quad \stackrel{t1\ ;\ t_2\ ;\ t_3\ ;\ t_4}{\Longrightarrow} \\ \quad\quad (\Pi_5^1,\ \ldots,\ (H_5^i, C_5^i,\ (E_5^n,\ ES_5^n,\ (l_{unit},\ p_{unit})\cdot RS_5^n) \uplus \overline{\overline{\mathcal{T}_5^i}}),\ \ldots \\ \quad\quad\quad (H^j,\ C^j,\ (E_c,\ \emptyset,\ \emptyset,\ \Lambda_c) \uplus \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi_5^k)\end{array}} \tag{49}$$

$$\frac{}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ (E^n,\ \emptyset,\ \emptyset,\ (l_{unit},\ p_{unit})) \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \stackrel{\epsilon}{\Longrightarrow} \\ \quad (\Pi_1^1,\ \ldots,\ (H_1^i,\ C_1^i,\ \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k)\end{array}} \tag{50}$$

Figure 26: Concurrency.

It therefore remains to provide a definition for the communication and concurrency primitives which do not appear in the sequential sub-language. Concurrency is introduced into the language by the rules in Figure 26. The $\mathbf{fork}(\Lambda)$ expression creates a new thread on the local process Π^i. The Λ expression is evaluated to obtain a closure $\langle\!\langle E_c,\ lv,\ \Lambda_c\rangle\!\rangle$. This closure is simply converted into a new thread $(E_c,\ \emptyset,\ \emptyset,\ \Lambda_c)$ and appended to the local multi-set of threads. The $\mathbf{rfork}(\Lambda_1,\ \Lambda_2)$ expression creates a new thread on a remote process Π^j. The expression Λ_1 is first evaluated to determine the remote process number j. Λ_2 is then evaluated to provide a closure, which is again converted to a thread, and appended to the multi-set of threads on process Π^j. When a thread has finished execution, it has the form: $(E^n,\ \emptyset,\ \emptyset,\ (l_{unit},\ p_{unit}))$. Such threads are identified by Rule 50 and removed from the multi-set of threads.

$$\frac{cn \text{ fresh}}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ C^i,\ (E^n,\ ES^n,\ RS^n,\ \mathbf{channel}()) \uplus \overline{\overline{\overline{\mathcal{T}^i}}}),\ \ldots,\ \Pi^k) \overset{\epsilon}{\Longrightarrow} \\ \quad (\Pi^1,\ \ldots,\ (H^i[l \mapsto cn][p \mapsto \text{t\_chan}],\ C^i, \\ \qquad (E^n[cn \mapsto (l,\ p)],\ ES^n,\ (l,\ p)\cdot RS^n) \uplus \overline{\overline{\overline{\mathcal{T}^i}}}),\ \ldots,\ \Pi^k)\end{array}} \tag{51}$$

$$\frac{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ C^i_1,\ (E^n_1,\ ES^n_1,\ RS^n_1,\ \Lambda_1) \uplus \overline{\overline{\overline{\mathcal{T}^i_1}}}),\ \ldots,\ \Pi^k_1) \overset{t_1}{\Longrightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (H^i_2,\ C^i_2,\ (E^n_2,\ ES^n_2,\ (l_1,\ p_1)\cdot RS^n_2) \uplus \overline{\overline{\overline{\mathcal{T}^i_2}}}),\ \ldots,\ \Pi^k_2) \\ (\Pi^1_2,\ \ldots,\ (H^i_2,\ C^i_2,\ (E^n_2,\ ES^n_2,\ RS^n_2,\ fetch(l_1,\ p_1)) \uplus \overline{\overline{\overline{\mathcal{T}^i_2}}}),\ \ldots,\ \Pi^k_2) \overset{t_2}{\Longrightarrow} \\ \quad (\Pi^1_3,\ \ldots,\ (H^i_3,\ C^i_3,\ (E^n_3,\ ES^n_3,\ RS^n_3) \uplus \overline{\overline{\overline{\mathcal{T}^i_3}}}),\ \ldots,\ \Pi^k_3) \\ (\Pi^1_3,\ \ldots,\ (H^i_3,\ C^i_3,\ (E^n_3,\ ES^n_3,\ RS^n_3,\ \Lambda_2) \uplus \overline{\overline{\overline{\mathcal{T}^i_3}}}),\ \ldots,\ \Pi^k_3) \overset{t_3}{\Longrightarrow} \\ \quad (\Pi^1_4,\ \ldots,\ (H^i_4,\ C^i_4,\ (E^n_4,\ ES^n_4,\ (l_2,\ p_2)\cdot(cn,\ ty)\cdot RS^n_4) \uplus \overline{\overline{\overline{\mathcal{T}^i_4}}}),\ \ldots,\ \Pi^k_4) \\ cn\ !\ (l_2,\ p_2)\end{array}}{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ C^i_1,\ (E^n_1,\ ES^n_1,\ RS^n_1,\ \mathbf{send}(\Lambda_1,\ \Lambda_2)) \uplus \overline{\overline{\overline{\mathcal{T}^i_1}}}),\ \ldots,\ \Pi^k_1) \\ \quad \xRightarrow{t_1\ ;\ t_2\ ;\ t_3\ ;\ cn\ !\ (l_2,\ p_2)} \\ \qquad (\Pi^1_4,\ \ldots,\ (H^i_4,\ C^i_4,\ (E^n_4,\ ES^n_4,\ (l_{unit},\ p_{unit})\cdot RS^n_4) \uplus \overline{\overline{\overline{\mathcal{T}^i_4}}}),\ \ldots,\ \Pi^k_4)\end{array}} \tag{52}$$

$$\frac{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ C^i_1,\ (E^n_1,\ ES^n_1,\ RS^n_1,\ \Lambda) \uplus \overline{\overline{\overline{\mathcal{T}^i_1}}}),\ \ldots,\ \Pi^k_1) \overset{t_1}{\Longrightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (H^i_2,\ C^i_2,\ (E^n_2,\ ES^n_2,\ (l_1,\ p_1)\cdot RS^n_2) \uplus \overline{\overline{\overline{\mathcal{T}^i_2}}}),\ \ldots,\ \Pi^k_2) \\ (\Pi^1_2,\ \ldots,\ (H^i_2,\ C^i_2,\ (E^n_2,\ ES^n_2,\ RS^n_2,\ fetch(l_1,\ p_1)) \uplus \overline{\overline{\overline{\mathcal{T}^i_2}}}),\ \ldots,\ \Pi^k_2) \overset{t_2}{\Longrightarrow} \\ \quad (\Pi^1_3,\ \ldots,\ (H^i_3,\ C^i_3,\ (E^n_3,\ ES^n_3,\ (cn,\ ty)\cdot RS^n_3) \uplus \overline{\overline{\overline{\mathcal{T}^i_3}}}),\ \ldots,\ \Pi^k_3) \\ cn\ ?\ (l_2,\ p_2)\end{array}}{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ C^i_1,\ (E^n_1,\ ES^n_1,\ RS^n_1,\ \mathbf{receive}(\Lambda)) \uplus \overline{\overline{\overline{\mathcal{T}^i_1}}}),\ \ldots,\ \Pi^k_1) \\ \quad \xRightarrow{t_1\ ;\ t_2\ ;\ cn\ ?\ (l_2,\ p_2)} \\ \qquad (\Pi^1_3,\ \ldots,\ (H^i_3,\ C^i_3,\ (E^n_3,\ ES^n_3,\ (l_2,\ p_2)\cdot RS^n_3) \uplus \overline{\overline{\overline{\mathcal{T}^i_3}}}),\ \ldots,\ \Pi^k_3)\end{array}} \tag{53}$$

Figure 27: Communication.

Communication between threads is performed across bi-directional channels. New channels are created by the **channel**() expression (Rule 51). With reference to Figure 20, the channel environment *NE* tracks the allocation of channel names. A channel is represented in the heap as a value *cn* of type t_chan.

Channels communicate pairs of value-locations and type-pointers $(l,\ p)$. These pairs are sent along channels by the $\mathbf{send}(\Lambda_1,\ \Lambda_2)$ expression (Rule 52) and received by the $\mathbf{receive}(\Lambda)$ expression (Rule 53). The channel along which communication takes place is provided by evaluating Λ_1 in the case of a send, or Λ in the case of receive. The second argument Λ_2 of a send is evaluated to provide the pair $(l,\ p)$. The communication operations are *blocking*. Both the sending and receiving threads are blocked until the pair $(l,\ p)$ is passed, which happens atomically. The notation for sending and receiving across channels is the same as the communication traces. For example, sending the pair $(l,\ p)$ across channel *cn* is written $cn\ !\ (l,\ p)$.

Concurrent ML:

```
let val c = channel()
    val _ = rfork(2, fn () => send(c, receive(c) + 3))
    val _ = send(c, 7)
in
    receive(c)
end
```

$\mathcal{DM}\Lambda$ **Translation:**

$$
\begin{array}{ll}
(\emptyset,\ \emptyset, & \\
\quad \textbf{let}\ (c,\ \text{t\_unit}) = \textbf{channel}\ () & \text{(i)} \\
\quad \textbf{in} & \\
\qquad \textbf{let}\ (\_,\ \text{t\_unit}) = & \\
\qquad\quad \textbf{rfork}\ (\textbf{scon}\ 2, & \\
\qquad\qquad \textbf{fn}\ (\_,\ \text{t\_unit} \rightarrow \text{t\_unit})\ = & \\
\qquad\qquad\quad \textbf{send}\ (\textbf{var}\ c,\ \textbf{prim}\ (\text{ADD}_i,\ (\textbf{receive}\ (\textbf{var}\ c),\ \textbf{scon}\ 3)))) & \text{(ii)} \\
\qquad \textbf{in} & \\
\qquad\quad \textbf{let}\ (\_,\ \text{t\_unit}) = \textbf{send}\ (\textbf{var}\ c,\ \textbf{scon}\ 7) & \text{(iii)} \\
\qquad\quad \textbf{in} & \\
\qquad\qquad \textbf{receive}(\textbf{var}\ c)) & \text{(iv)}
\end{array}
$$

Simplified Evaluation Trace:

$$
\begin{array}{ll}
\text{(i)} & \epsilon \\
\text{(ii)} & \epsilon \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) \\
\text{(iii)} & (\epsilon\ ;\ c\ !\ (l_1,\ p_1)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) \\
\text{(iv)} & (\epsilon\ ;\ c\ !\ (l_1,\ p_1)\ ;\ c\ ?\ (l_2,\ p_2)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2))
\end{array}
$$

Trace Reduction:

$$
\begin{array}{lll}
 & (\epsilon\ ;\ c\ !\ (l_1,\ p_1)\ ;\ c\ ?\ (l_2,\ p_2)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) & \\
= & (c\ !\ (l_1,\ p_1)\ ;\ c\ ?\ (l_2,\ p_2)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) & \text{by Rule 27} \\
= & c\ ?\ (l_2,\ p_2) \parallel c\ !\ (l_2,\ p_2) & \text{by Rule 31} \\
= & c\ !\ (l_2,\ p_2) \parallel c\ ?\ (l_2,\ p_2) & \text{by Rule 29} \\
= & \epsilon & \text{by Rule 30}
\end{array}
$$

Figure 28: Trace Reduction Example.

Earlier in this section (4.1) it was stated that the traces for a correct program should always reduce to ϵ. Having now provided the transitions for the communication and concurrency primitives, Figure 28 shows that this is the case for the example given in the introduction. For clarity, the traces have been simplified by removing all the ϵ traces resulting from the intermediate steps in the evaluation. The first pair that is transmitted across the channel c, corresponding to the special constant 7, is denoted $(l_1,\ p_1)$. The second pair, corresponding to the special constant 10 (7+3), is denoted $(l_2,\ p_2)$.

4.4 Distributed Garbage Collection

In this section an abstract machine is presented which describes the distributed garbage collection algorithm. The syntax of this abstract machine, called the $\mathcal{DGC}$ machine, is given in Figure 29.

Machine State	GC	::=	$\overline{\pi}^{\,k}$
GC Process	π	::=	$(Hf,\ Ht,\ PF,\ LF,\ \overline{\overline{T}})$
GC Thread	T	::=	$E \mid S \mid cn_l \mid cn_p \mid (p_1,\ p_2) \mid (l_1,\ p_1,\ l_2)$
Trace	t	::=	$\epsilon \mid t_1 \,;\, t_2 \mid t_1 \parallel t_2 \mid cn\,!\,m \mid cn\,?\,m$
Message	m	::=	$(p) \mid (l,\ p) \mid (E)$

Figure 29: $\mathcal{DGC}$ Abstract Machine Syntax.

The organisation of $\mathcal{DGC}$ machine is very similar to the $\mathcal{DM}\Lambda$ machine. There are a fixed number k of concurrently executing processes π each executing a multi-set of threads T. This similarity is intended. There is a one-to-one correspondence between $\mathcal{DGC}$ processes and $\mathcal{DM}\Lambda$ processes. In an actual implementation the two machines would be combined. The combined machine would alternate between the execution of $\mathcal{DM}\Lambda$ and $\mathcal{DGC}$ processes and threads during cycles of program execution and garbage collection. Figure 30 defines the parallel execution of garbage collection processes and threads.

$$\frac{\pi_1^1 \overset{t_1}{\Longrightarrow}_{gc} \pi_2^1 \quad \cdots \quad \pi_1^k \overset{t_k}{\Longrightarrow}_{gc} \pi_2^k}{(\pi_1^1,\ \ldots,\ \pi_1^k) \overset{t_1 \parallel \cdots \parallel t_k}{\Longrightarrow}_{gc} (\pi_2^1,\ \ldots,\ \pi_2^k)} \tag{54}$$

$$\frac{T_1 \overset{t_1}{\Longrightarrow}_{gc} T_3 \qquad \overline{\overline{T_2}} \overset{t_2}{\Longrightarrow}_{gc} \overline{\overline{T_4}}}{T_1 \uplus \overline{\overline{T_2}} \overset{t_1 \parallel t_2}{\Longrightarrow}_{gc} T_3 \uplus \overline{\overline{T_4}}} \tag{55}$$

Figure 30: Execution of $\mathcal{DGC}$ Processes and Threads.

Each $\mathcal{DGC}$ process is effectively a separate copy of the uniprocessor algorithm. Therefore, each process contains a copy of the GC state as defined in Figure 10. There are separate threads for garbage collecting environments, stacks, values, and types. Environments are garbage collected using E threads and stacks are garbage collected using S threads. The garbage collection of types and values is more complex. Recall from the description of the distributed garbage collection algorithm, at the beginning of this section, that each process is responsible for garbage collecting data that is contained within its own heap. Thus, in the $\mathcal{DGC}$ machine a mechanism is required for communicating pointers to the appropriate processes for collection. This is achieved through the use of *server* threads. Each $\mathcal{DGC}$ process has two server threads: one for types and one for values. Each of these server threads has an associated channel along which messages are sent and received. Hence, the server threads are named cn_l^i and cn_p^i, where $0 < i \leq k$. When an external pointer is encountered during garbage collection, it is simply sent along a channel to the appropriate server thread which returns an updated pointer along the same channel.

The server thread is determined using the num operation defined in Section 4.2. For example, a type referenced by the pointer p_1 is garbage collected by sending it to a server, where $i = num(p_1)$: $cn_p^i\ !\ (p_1)$. The updated pointer p_2 is subsequently retrieved from the same server: $cn_p^i\ ?\ (p_2)$. For convenience, these operations are combined in the garbage collection rules e.g. $cn_p^i\ !\ (p_1)\ ?\ (p_2)$, which is shorthand for $cn_p^i\ !\ (p_1)\ ;\ cn_p^i\ ?\ (p_2)$. In order to simplify the garbage collection rules, a distinction is not made between local and remote data. All type-pointers and value-locations are sent in this manner to the servers, even if the server is on the same process. It would be straightforward to optimise the local case, but the number of rules would double.

It may be the case that two (or more) threads attempt to communicate with a single server thread at the same time. In this case, the server makes a non-deterministic choice between the threads. Only one thread is permitted to communicate with the server. The remaining threads are blocked until the server becomes available. Communication with the server is always done by performing a send operation followed by a receive. This ordering ensures that the correct thread receives the reply from the server.

The server threads do not actually perform garbage collection. A copying operation may require the co-operation of a number of other servers. This could easily lead to a deadlock owing to the blocking nature of the communication channel, e.g. two servers may become blocked waiting for each other to finish. One solution would be to fork a new server thread every time the server becomes busy. However, each of these server threads would require a separate communication channel which would considerably complicate the collection algorithm.

The solution adopted here involves the use of *worker* threads to perform the copying operation. Recall from Section 4.1 that a pointer may be reserved on the heap. Thus, the server thread simply reserves and returns a pointer while a separate worker thread is forked to copy the data. A worker thread for copying a value is denoted $(l_1,\ p_1,\ l_2)$. The value referenced by l_1 is copied recursively into the location referenced by l_2 using the type reference by p_1. Similarly, a worker thread for copying a type is denoted $(p_1,\ p_2)$. The type referenced by p_1 is copied recursively into the heap referenced by p_2.

The rule in Figure 32 illustrates how distributed garbage collection is combined with the rules of the $\mathcal{D}M\Lambda$ abstract machine. This rule generates a single $\mathcal{D}GC$ process from a single $\mathcal{D}M\Lambda$ process. Distributed garbage collection requires this operation to be performed on every process in the $\mathcal{D}M\Lambda$ machine. The set of garbage collection threads (roots) for a single $\mathcal{D}GC$ process is generated from the union of all the environments and stacks contained within the threads of a $\mathcal{D}M\Lambda$ process. The set of threads also includes the value and type servers for processing external references.

Garbage collection proceeds until all of the roots have been processed. The set of $\mathcal{D}M\Lambda$ threads is then rebuilt with the new environments and stacks and normal evaluation is resumed. Note that the contents of the cache is cleared by garbage collection.

An example distributed garbage collection with two processes is illustrated in Figure 31. In the example, an integer list type is garbage collected. Initially, the list type *t_list* is referenced by the type-pointer p_1 and is contained within the *from* heap of process π^1. The list type contains a pointer p_2 to the integer type *t_int* contained within the *from* heap of process π^2.

Garbage collection is initiated by sending the pointer p_1 to the server thread cn_p^1 on process π^1. The server thread reserves a pointer p_3 on the *to* heap of process π^1 and returns this pointer. A worker thread $(p_1,\ p_3)$ is created to perform the copy of the list type. During the copy operation, the worker thread encounters the pointer p_2 to the integer type. This pointer is sent to the server thread cn_p^2 on process π^2. The server thread subsequently reserves a pointer p_4 on the *to* heap of process π^2 and returns this to the worker thread. Now a new worker thread $(p_2,\ p_4)$ is created on process π^2 to perform the copy of the integer type. Meanwhile, the worker thread on process π^1 completes the copy of the list type with the updated pointer p_4. Once the worker thread on process π^2 has completed the copy of the integer type, the garbage collection is complete. The type-pointer p_3 references the garbage collected type in the *to* heap of process π^1.

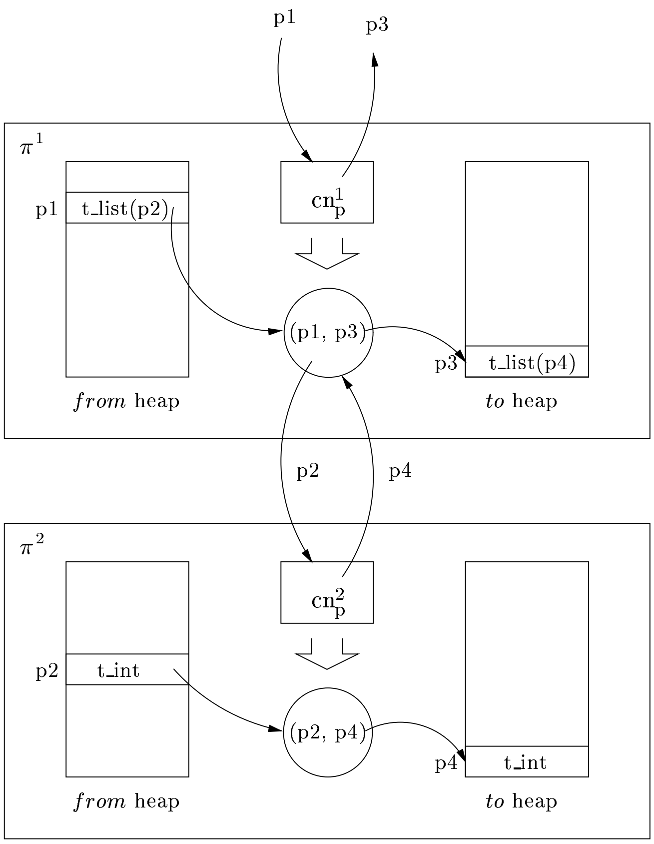

Figure 31: Distributed Garbage Collection Example.

$$
\frac{\begin{array}{l}\overline{\overline{\mathcal{T}_1^i}} = \{(E_1^1,\ ES_1^1,\ RS_1^1),\ \ldots,\ (E_1^j,\ ES_1^j,\ RS_1^j)\} \\ (\pi_1^1,\ \ldots,\ (H_1^i,\ \emptyset,\ \emptyset,\ \emptyset,\ E_1^1 \uplus ES_1^1 \uplus RS_1^1 \uplus \cdots \\ \quad \cdots \uplus E_1^j \uplus ES_1^j \uplus RS_1^j \uplus cn_l \uplus cn_p),\ \ldots,\ \pi_1^k) \stackrel{t}{\Longrightarrow}_{gc} \\ \qquad (\pi_2^1,\ \ldots,\ (H_1^i,\ H_2^i,\ PF^i,\ LF^i,\ E_2^1 \uplus ES_2^1 \uplus RS_2^1 \uplus \cdots \\ \qquad \quad \cdots \uplus E_2^j \uplus ES_2^j \uplus RS_2^j \uplus cn_l \uplus cn_p),\ \ldots,\ \pi_2^k) \\ \overline{\overline{\mathcal{T}_2^i}} = \{(E_2^1,\ ES_2^1,\ RS_2^1),\ \ldots,\ (E_2^j,\ ES_2^j,\ RS_2^j)\}\end{array}}{(\Pi^1,\ \ldots,\ (H_1^i,\ C_1^i,\ \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi^k) \stackrel{t}{\Longrightarrow} (\Pi^1,\ \ldots,\ (H_2^i,\ \emptyset,\ \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi^k)} \tag{56}
$$

Figure 32: Distributed Garbage Collection Introduction.

Figures 33 through 37 define the behaviour of the garbage collection threads. It is worth noting that these rules are very similar to their sequential equivalents in Figures 12 through 15. Where the sequential algorithm collects a pointer $(Hf,\ Ht_1,\ PF_1,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ p_2)$, the distributed algorithm communicates with a server $cn_p^{num(p_1)}\ !\ (p_1)\ ?\ (p_2)$. A sequential collection effectively corresponds to a single garbage collection process collecting a single $\mathcal{DM}\Lambda$ process in the $\mathcal{DGC}$ machine.

The server thread for types cn_p is defined in Figure 33. In Rule 57, the type referenced by p_1 has already been collected. The entry in the forwarding table $PF(p_1)$ is therefore returned. In Rule 58 the type has not been collected. A pointer p_2 is reserved in Ht, to hold the collected type, and returned. A worker thread $(p_1,\ p_2)$ is created to copy the type, and a mapping $p_1 \mapsto p_2$ is added to the forwarding table. Note that the server is still present in the set of threads at the end of the rule. This has the effect of restarting the server. The server thread for values cn_l is defined in Figure 35.

The worker thread for types $(p_1,\ p_2)$ is defined in Figure 34. There are separate rules for each of the types in the language. Any type pointers that are encountered are collected by sending them to the appropriate type server. The heap is accessed directly, instead of going via the $fetch$ operation, as the types will always be contained within the local heap. The worker thread for values $(l_1,\ p_1,\ l_2)$ is defined in Figure 35. There are separate rules for each of the values in the language. The type information referenced by p_1 is used to guide the collection, but the type itself is not collected by these rules. Any value locations that are encountered are sent to the appropriate value server.

The thread for collecting environments is given in Figure 37. The environment E_1 is decomposed into a constructor environment CE_1, and a value environment VE_1. A new constructor environment CE_2 is built by collecting all of the types referenced in CE_1. This is achieved by sending all of the type references to the type servers. Similarly, a new value environment VE_2 is built by collecting all of the values and types referenced in VE_1. This is achieved by sending all of the references to the type and value servers. A new environment E_2 is built from CE_2, and VE_2.

The final Figure 38 contains the rules for garbage collecting the stacks ES and RS. There are separate rules depending on wether the item at the top of the stack is a type (Rule 74), a value (Rule 75), or an environment (Rule 76). The base case, an empty stack, is handled by Rule 73.

$$\frac{t_1 = cn_p^i\ ?\ (p_1) \qquad p_1 \in \mathrm{Dom}\ PF^i \qquad p_2 = PF^i(p_1) \qquad t_2 = cn_p^i\ !\ (p_2)}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (cn_p^i) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (cn_p^i) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{57}$$

$$\frac{t_1 = cn_p^i\ ?\ (p_1) \qquad p_1 \notin \mathrm{Dom}\ PF^i \qquad Ht^i \uparrow p_2 \qquad t_2 = cn_p^i\ !\ (p_2)}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (cn_p^i) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i[p_1 \mapsto p_2],\ LF^i,\ (cn_p^i) \uplus (p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{58}$$

Figure 33: Type Server Thread.

$$\frac{Hf^i(p_1) = tn}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2 \mapsto tn],\ PF^i,\ LF^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{59}$$

$$\frac{Hf^i(p_1) = tn(p_3) \qquad t = cn_p^{num(p_3)}\ !\ (p_3)\ ?\ (p_4)}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t}{\Longrightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2 \mapsto tn(p_4)],\ PF^i,\ LF^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{60}$$

$$\frac{Hf^i(p_1) = \overline{p_3}^{\,n} \qquad t = cn_p^{num(p_3^1)}\ !\ (p_3^1)\ ?\ (p_4^1)\ ;\ \cdots\ ;\ cn_p^{num(p_3^n)}\ !\ (p_3^n)\ ?\ (p_4^n)}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t}{\Longrightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2 \mapsto \overline{p_4}^{\,n}],\ PF^i,\ LF^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{61}$$

$$\frac{Hf^i(p_1) = p_3 \rightarrow p_4 \qquad t = cn_p^{num(p_3)}\ !\ (p_3)\ ?\ (p_5)\ ;\ cn_p^{num(p_4)}\ !\ (p_4)\ ?\ (p_6)}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t}{\Longrightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2 \mapsto p_5 \rightarrow p_6],\ PF^i,\ LF^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{62}$$

$$\frac{Hf^i(p_1) = tv}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2 \mapsto tv],\ PF^i,\ LF^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{63}$$

$$\frac{\begin{array}{l} Hf^i(p_1) = \forall\ \overline{p_3}^{\,n}.\ p_4 \\ t_1 = cn_p^{num(p_3^1)}\ !\ (p_3^1)\ ?\ (p_5^1)\ ;\ \cdots\ ;\ cn_p^{num(p_3^n)}\ !\ (p_3^n)\ ?\ (p_5^n) \\ t_2 = cn_p^{num(p_3)}\ !\ (p_4)\ ?\ (p_6) \end{array}}{(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{gc} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_1 \mapsto \forall\ \overline{p_5}^{\,n}.\ p_6],\ PF^i,\ LF^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)} \tag{64}$$

Figure 34: Type Worker Thread.

$$\frac{t_1 = cn_l^i \; ? \; (l_1,\; p_1) \qquad l_1 \in \mathrm{Dom}\; LF^i \qquad l_2 = LF^i(l_1) \qquad t_2 = cn_l^i \; ! \; (l_2,\; p_1)}{\begin{array}{l}(\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i,\; LF^i,\; (cn_l^i) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k) \xRightarrow{t_1\;;\;t_2}_{gc} \\ \quad (\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i[p_1 \;\mapsto\; p_2],\; LF^i,\; (cn_l^i) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k)\end{array}} \tag{65}$$

$$\frac{t_1 = cn_l^i \; ? \; (l_1,\; p_1) \qquad l_1 \notin \mathrm{Dom}\; LF^1 \qquad Ht^i \uparrow l_2 \qquad t_2 = cn_l^i \; ! \; (l_2,\; p_1)}{\begin{array}{l}(\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i,\; LF^i,\; (cn_l^i) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k) \xRightarrow{t_1\;;\;t_2}_{gc} \\ \quad (\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i,\; LF^i[l_1 \;\mapsto\; l_2],\; (cn_l^i) \uplus (l_1,\; p_1,\; l_2) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k)\end{array}} \tag{66}$$

Figure 35: Value Server Thread.

$$\frac{Hf^i(p_1) = tn}{\begin{array}{l}(\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i,\; LF^i,\; (l_1,\; p_1,\; l_2) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k) \xRightarrow{\epsilon}_{gc} \\ \quad (\pi^1,\; \ldots,\; (Hf^i,\; Ht^i[l_2 \;\mapsto\; Hf^i(l_1)],\; PF^i,\; LF^i,\; \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k)\end{array}} \tag{67}$$

$$\frac{Hf^i(p_1) = tn(p_2) \qquad Hf^i(l_1) = con(l_3) \qquad t = cn_l^{num(l_3)} \; ! \; (l_3,\; p_2) \; ? \; (l_4,\; p_2)}{\begin{array}{l}(\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i,\; LF^i,\; (l_1,\; p_1,\; l_2) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k) \xRightarrow{t}_{gc} \\ \quad (\pi^1,\; \ldots,\; (Hf^i,\; Ht^i[l_2 \;\mapsto\; con(l_4)],\; PF^i,\; LF^i,\; \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k)\end{array}} \tag{68}$$

$$\frac{\begin{array}{l}Hf^i(p_1) = \overline{p_2}^{\,n} \qquad Hf^i(l_1) = \overline{l_3}^{\,n} \\ t = cn_l^{num(l_3^1)} \; ! \; (l_3^1,\; p_2^1) \; ? \; (l_4^1,\; p_2^1) \; ; \; \cdots \; ; \; cn_p^{num(l_3^n)} \; ! \; (l_3^n,\; p_2^n) \; ? \; (l_4^n,\; p_2^n)\end{array}}{\begin{array}{l}(\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i,\; LF^i,\; (l_1,\; p_1,\; l_2) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k) \xRightarrow{t}_{gc} \\ \quad (\pi^1,\; \ldots,\; (Hf^i,\; Ht^i[l_2 \;\mapsto\; (l_4^1,\; \ldots,\; l_4^n)],\; PF^i,\; LF^i,\; \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k)\end{array}} \tag{69}$$

$$\frac{\begin{array}{l}Hf^i(p_1) = p_2 \rightarrow p_3 \qquad Hf^i(l_1) = \langle\!\langle E_1,\; \overline{lv}^{\,l},\; \Lambda \rangle\!\rangle \\ (\pi_1^1,\; \ldots,\; (Hf_1^i,\; Ht_1^i,\; PF_1^i,\; LF_1^i,\; E_1 \uplus \overline{\overline{\overline{T_1^i}}}),\; \ldots,\; \pi_1^k) \xRightarrow{t}_{gc} \\ \quad (\pi_2^1,\; \ldots,\; (Hf_2^i,\; Ht_2^i,\; PF_2^i,\; LF_2^i,\; E_2 \uplus \overline{\overline{\overline{T_2^i}}}),\; \ldots,\; \pi_2^k)\end{array}}{\begin{array}{l}(\pi_1^1,\; \ldots,\; (Hf_1^i,\; Ht_1^i,\; PF_1^i,\; LF_1^i,\; (l_1,\; p_1,\; l_2) \uplus \overline{\overline{\overline{T_1^i}}}),\; \ldots,\; \pi_1^k) \xRightarrow{t}_{gc} \\ \quad (\pi_2^1,\; \ldots,\; (Hf_2^i,\; Ht_2^i[l_2 \;\mapsto\; \langle\!\langle E_2,\; \overline{lv}^{\,l},\; \Lambda \rangle\!\rangle],\; PF_2^i,\; LF_2^i,\; \overline{\overline{\overline{T_2^i}}}),\; \ldots,\; \pi_2^k)\end{array}} \tag{70}$$

$$\frac{Hf^i(p_1) = p_2 \rightarrow p_3 \qquad Hf^i(l_1) = \Omega}{\begin{array}{l}(\pi^1,\; \ldots,\; (Hf^i,\; Ht^i,\; PF^i,\; LF^i,\; (l_1,\; p_1,\; l_2) \uplus \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k) \xRightarrow{\epsilon}_{gc} \\ \quad (\pi^1,\; \ldots,\; (Hf^i,\; Ht^i[l_2 \;\mapsto\; \Omega],\; PF^i,\; LF^i,\; \overline{\overline{\overline{T^i}}}),\; \ldots,\; \pi^k)\end{array}} \tag{71}$$

Figure 36: Value Worker Thread.

$$
\frac{\begin{array}{l}
E_1 = (TE,\ NE,\ CE_1,\ VE_1) \\
CE_1 = \{con^1 \ \mapsto\ p_1^1,\ \ldots,\ con^k \ \mapsto\ p_1^k\} \\
t_1 = cn_p^{num(p_1^1)}\ !\ (p_1^1)\ ?\ (p_2^1)\ ;\ \cdots\ ;\ cn_p^{num(p_1^k)}\ !\ (p_1^k)\ ?\ (p_2^k) \\
CE_2 = \{con^1 \ \mapsto\ p_2^1,\ \ldots,\ con^k \ \mapsto\ p_2^k\} \\
VE_1 = \{lv^1 \ \mapsto\ (l_1^1,\ p_3^1),\ \ldots,\ lv^l \ \mapsto\ (l_1^l,\ p_5^l)\} \\
t_2 = cn_p^{num(p_3^1)}\ !\ (p_3^1)\ ?\ (p_4^1)\ ;\ \cdots\ ;\ cn_p^{num(p_3^l)}\ !\ (p_3^l)\ ?\ (p_4^l) \\
t_3 = cn_l^{num(l_1^1)}\ !\ (l_1^1,\ p_4^1)\ ?\ (l_2^1,\ p_4^1)\ ;\ \cdots\ ;\ cn_l^{num(l_1^l)}\ !\ (l_1^l,\ p_4^l)\ ?\ (l_2^l,\ p_4^l) \\
VE_2 = \{lv^1 \ \mapsto\ (l_2^1,\ p_4^1),\ \ldots,\ lv^l \ \mapsto\ (l_2^l,\ p_4^l)\} \\
E_2 = (TE,\ NE,\ CE_2,\ VE_2)
\end{array}}{\begin{array}{l}
(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (E_1) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \xRightarrow{t_1\ ;\ t_2\ ;\ t_3}_{gc} \\
\quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ (E_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)
\end{array}} \tag{72}
$$

Figure 37: Environment Thread.

$$
\frac{}{\begin{array}{l}
(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ () \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \xRightarrow{\epsilon}_{gc} \\
\quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ PF^i,\ LF^i,\ () \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)
\end{array}} \tag{73}
$$

$$
\frac{\begin{array}{l}
t_1 = cn_p^{num(p_1)}\ !\ (p_1)\ ?\ (p_2) \\
(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ PF_1^i,\ LF_1^i,\ (S_1) \uplus \overline{\overline{\overline{T_1^i}}}),\ \ldots,\ \pi_1^k) \xRightarrow{t_2}_{gc} \\
\quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ PF_2^i,\ LF_2^i,\ (S_2) \uplus \overline{\overline{\overline{T_2^i}}}),\ \ldots,\ \pi_2^k)
\end{array}}{\begin{array}{l}
(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ PF_1^i,\ LF_1^i,\ (p_1{\cdot}S_1) \uplus \overline{\overline{\overline{T_1^i}}}),\ \ldots,\ \pi_1^k) \xRightarrow{t_1\ ;\ t_2}_{gc} \\
\quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ PF_2^i,\ LF_2^i,\ (p_2{\cdot}S_2) \uplus \overline{\overline{\overline{T_2^i}}}),\ \ldots,\ \pi_2^k)
\end{array}} \tag{74}
$$

$$
\frac{\begin{array}{l}
t_1 = cn_p^{num(p_1)}\ !\ (p_1)\ ?\ (p_2) \qquad t_2 = cn_l^{num(l_1)}\ !\ (l_1,\ p_1)\ ?\ (l_2,\ p_1) \\
(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ PF_1^i,\ LF_1^i,\ (S_1) \uplus \overline{\overline{\overline{T_1^i}}}),\ \ldots,\ \pi_1^k) \xRightarrow{t_3}_{gc} \\
\quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ PF_2^i,\ LF_2^i,\ (S_2) \uplus \overline{\overline{\overline{T_2^i}}}),\ \ldots,\ \pi_2^k)
\end{array}}{\begin{array}{l}
(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ PF_1^i,\ LF_1^i,\ ((l_1,\ p_1){\cdot}S_1) \uplus \overline{\overline{\overline{T_1^i}}}),\ \ldots,\ \pi_1^k) \xRightarrow{t_1\ ;\ t_2\ ;\ t_3}_{gc} \\
\quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ PF_2^i,\ LF_2^i,\ ((l_2,\ p_2){\cdot}S_2) \uplus \overline{\overline{\overline{T_2^i}}}),\ \ldots,\ \pi_2^k)
\end{array}} \tag{75}
$$

$$
\frac{\begin{array}{l}
(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ PF_1^i,\ LF_1^i,\ (E_1) \uplus \overline{\overline{\overline{T_1^i}}}),\ \ldots,\ \pi_1^k) \xRightarrow{t_1}_{gc} \\
\quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ PF_2^i,\ LF_2^i,\ (E_2) \uplus \overline{\overline{\overline{T_2^i}}}),\ \ldots,\ \pi_2^k) \\
(\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ PF_2^i,\ LF_2^i,\ (S_1) \uplus \overline{\overline{\overline{T_2^i}}}),\ \ldots,\ \pi_2^k) \xRightarrow{t_2}_{gc} \\
\quad (\pi_3^1,\ \ldots,\ (Hf_3^i,\ Ht_3^i,\ PF_3^i,\ LF_3^i,\ (S_2) \uplus \overline{\overline{\overline{T_3^i}}}),\ \ldots,\ \pi_3^k)
\end{array}}{\begin{array}{l}
(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ PF_1^i,\ LF_1^i,\ (E_1{\cdot}S_1) \uplus \overline{\overline{\overline{T_1^i}}}),\ \ldots,\ \pi_1^k) \xRightarrow{t_1\ ;\ t_2}_{gc} \\
\quad (\pi_3^1,\ \ldots,\ (Hf_3^i,\ Ht_3^i,\ PF_3^i,\ LF_3^i,\ (E_2{\cdot}S_2) \uplus \overline{\overline{\overline{T_3^i}}}),\ \ldots,\ \pi_3^k)
\end{array}} \tag{76}
$$

Figure 38: Stack Thread.

5 Further Work and Concluding Remarks

Modern compilers for higher-order typed programming languages use typed intermediate languages to structure the compilation process. The use of such languages in this report has allowed the construction of efficient tag-free algorithms for both the sequential and distributed cases. In addition, the use of abstract machine notation has enabled a formal presentation of memory management separately from other aspects such as syntax and type-correctness. Unlike traditional models, the abstract machine exposes many important details such as the heap, stacks, and environments and provides an implementor with an unambiguous and precise description of memory management.

The use of type information also enables a number of extensions to the garbage collection algorithm. The model presented here only deals with heap garbage collection. However, space leaks in the heap may also result from stack and environment garbage. Stack garbage results from recursive tail-calls, and environment garbage results from unused bindings. As an example, Figure 39 illustrates the removal of unused bindings from closure environments. CON returns the set of constructors which appear in Λ, and VAR returns the set of variables which appear in Λ. Rule 77 is for the sequential case, and Rule 78 is for the distributed case. These rules could easily be combined with Rule 20 (sequential) and Rule 70 (distributed) to improve the efficiency of the garbage collection algorithms. The collection of stack garbage is a more complex problem and is currently under investigation.

$$\frac{CON(\Lambda) \subseteq \mathrm{Dom}\ CE^1 \qquad VAR(\Lambda) \subseteq \mathrm{Dom}\ VE^1}{\begin{array}{l}\langle\!\langle (TE,\ CE^1 \uplus CE^2,\ VE^1 \uplus VE^2),\ \overline{lv}^{\,k},\ \Lambda \rangle\!\rangle \Rightarrow_{gc} \\ \quad \langle\!\langle (TE,\ CE^1,\ VE^1),\ \overline{lv}^{\,k},\ \Lambda \rangle\!\rangle\end{array}} \tag{77}$$

$$\frac{CON(\Lambda) \subseteq \mathrm{Dom}\ CE^1 \qquad VAR(\Lambda) \subseteq \mathrm{Dom}\ VE^1}{\begin{array}{l}\langle\!\langle (TE,\ NE,\ CE^1 \uplus CE^2,\ VE^1 \uplus VE^2),\ \overline{lv}^{\,k},\ \Lambda \rangle\!\rangle \Rightarrow_{gc} \\ \quad \langle\!\langle (TE,\ NE,\ CE^1,\ VE^1),\ \overline{lv}^{\,k},\ \Lambda \rangle\!\rangle\end{array}} \tag{78}$$

Figure 39: Closure Garbage Collection.

The abstract machine model has also provided a definition for the mechanisms of the LEMMA memory interface. In order to clearly present the memory management operations, a number of simplifications have been made in the distributed case, e.g. mutable data is not cached. An obvious extension would be an inclusion of the optimisation techniques described in [12] and [14]. Beyond these optimisations, it would also be interesting to adapt the model to cope with wide-area distribution. In this setting, the intention would clearly be to minimise the amount of communication. Also, the model of distributed shared memory may be unrealistic and require a change to a purely message-passing paradigm. It would also be necessary to cope with the presence of computation and communication failures. This may require some changes to the Concurrent ML primitives. For example, the addition of time-outs to the communication operations.

A Sequential Abstract Machine Definition

A.1 Programs

$$
\frac{\begin{array}{l}
P = (\{D^1,\ \ldots,\ D^k\},\ \{X^1,\ \ldots,\ X^l\},\ \Lambda) \\
(H_1,\ E_1,\ ES_1,\ RS_1,\ D^1) \Rightarrow (H_2,\ E_2,\ ES_1,\ RS_1)\ \cdots \\
\quad (H_k,\ E_k,\ ES_1,\ RS_1,\ D^k) \Rightarrow (H_{k+1},\ E_{k+1},\ ES_1,\ RS_1) \\
(H_{k+1},\ E_{k+1},\ ES_1,\ RS_1,\ X^1) \Rightarrow (H_{k+2},\ E_{k+2},\ ES_1,\ RS_1)\ \cdots \\
\quad (H_{k+l},\ E_{k+l},\ ES_1,\ RS_1,\ X^l) \Rightarrow (H_{k+l+1},\ E_{k+l+1},\ ES_1,\ RS_1) \\
(H_{k+l+1},\ E_{k+l+1},\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_{k+l+2},\ E_{k+l+2},\ ES_2,\ RS_2)
\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ P) \Rightarrow (H_{k+l+2},\ E_{k+l+2},\ ES_2,\ RS_2)} \tag{79}
$$

Comment: (Rule 79) The Program P is decomposed into datatypes D, exceptions X, and an expression Λ. These components are then evaluated in turn by sequences of transitions.

A.2 Datatype and Exception Declarations

$$
\frac{}{\begin{array}{l}
(H,\ E,\ ES,\ RS,\ \textbf{datatype}\ tn\ \textbf{of}\ \overline{(con,\ \tau)}^{\,k}) \Rightarrow \\
\quad (H[p_1 \mapsto \tau^1 \to tn,\ \ldots,\ p_k \mapsto \tau^k \to tn], \\
\quad\ E[tn][con^1 \mapsto p_1,\ \ldots,\ con^k \mapsto p_k],\ ES,\ RS)
\end{array}} \tag{80}
$$

Comment: (Rule 80) Datatype constructors are represented as functions from the constructor argument type to the datatype name $\tau \to tn$. The type of a nullary constructor is t_unit $\to tn$. These function types are allocated on the type heap $H[p \mapsto \tau \to tn]$, and entered into the environment $E[tn][con \mapsto p]$.

$$
\frac{}{\begin{array}{l}
(H,\ E,\ ES,\ RS,\ \textbf{datatype}\ (\overline{tv}^{\,k},\ tn)\ \textbf{of}\ \overline{(con,\ \tau)}^{\,l}) \Rightarrow \\
\quad (H[p_1 \mapsto \forall\,\overline{tv}^{\,k}.\,\tau^1 \to tn,\ \ldots,\ p_l \mapsto \forall\,\overline{tv}^{\,k}.\,\tau^l \to tn], \\
\quad\ E[tn][con^1 \mapsto p_1,\ \ldots,\ con^l \mapsto p_l],\ ES,\ RS)
\end{array}} \tag{81}
$$

Comment: (Rule 81) Polymorphic datatype constructors are represented by functions $\forall\,\overline{tv}^{\,k}.\,\tau \to tn$.

$$
\frac{}{\begin{array}{l}
(H,\ E,\ ES,\ RS,\ \textbf{exception}\ (con,\ \tau)) \Rightarrow \\
\quad (H[p \mapsto \tau \to \mathrm{t\_exn}],\ E[con \mapsto p],\ ES,\ RS)
\end{array}} \tag{82}
$$

Comment: (Rule 82) The effect of an exception declaration is analogous to that of adding a constructor to a pre-defined datatype named t_exn.

A.3 Values

$$\frac{}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \textbf{scon}\ scon) \Rightarrow \\ \quad (H[l\ \mapsto\ scon][p\ \mapsto\ \tau_{scon}],\ E,\ ES,\ (l,\ p){\cdot}RS)\end{array}} \tag{83}$$

Comment: (Rule 83) τ_{scon} is the type of the special constant *scon* (e.g. t_int).

$$\frac{E(lv) = (l,\ p)}{(H,\ E,\ ES,\ RS,\ \textbf{var}\ lv) \Rightarrow (H,\ E,\ ES,\ (l,\ p){\cdot}RS)} \tag{84}$$

$$\frac{\begin{array}{l}E(lv) = (l_1,\ p_1) \\ (H_1[p_2\ \mapsto\ \overline{\tau}^{\,k}],\ E,\ ES,\ RS,\ instance(p_1,\ p_2)) \Rightarrow (H_2,\ E,\ ES,\ p_3{\cdot}RS)\end{array}}{(H_1,\ E,\ ES,\ RS,\ \textbf{var}\ (\overline{\tau}^{\,k},\ lv)) \Rightarrow (H_2,\ E,\ ES,\ (l_1,\ p_3){\cdot}RS)} \tag{85}$$

$$\frac{}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \textbf{fn}\ (lv,\ \tau_1 \rightarrow \tau_2) = \Lambda) \Rightarrow \\ \quad (H[l\ \mapsto\ \langle\!\langle E,\ lv,\ \Lambda\rangle\!\rangle][p\ \mapsto\ \tau_1 \rightarrow \tau_2],\ E,\ ES,\ (l,\ p){\cdot}RS)\end{array}} \tag{86}$$

$$\frac{}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \textbf{fn}\ (\overline{lv}^{\,k},\ \overline{\tau_1}^{\,k} \rightarrow \tau_2) = \Lambda) \Rightarrow \\ \quad (H[l\ \mapsto\ \langle\!\langle E,\ \overline{lv}^{\,k},\ \Lambda\rangle\!\rangle][p\ \mapsto\ \overline{\tau_1}^{\,k} \rightarrow \tau_2],\ E,\ ES,\ (l,\ p){\cdot}RS)\end{array}} \tag{87}$$

Comment: (Rules 86 and 87) These rules allocate a new closure on the value heap. The second rule is for functions which take multiple arguments. The closure consists of a copy of the environment, a variables (or list of variables) to be bound to the function parameters, and an expression for the body of the function.

A.4 Constructors

$$\frac{E(con) = p_1 \qquad H(p_1) = p_2 \rightarrow p_3}{(H,\ E,\ ES,\ RS,\ \textbf{con}\ con) \Rightarrow (H[l_1\ \mapsto\ con],\ E,\ ES,\ (l_1,\ p_3){\cdot}RS)} \tag{88}$$

$$\frac{\begin{array}{l}E(con) = p_1 \\ (H_1[p_2\ \mapsto\ \overline{\tau}^{\,k}],\ E,\ ES,\ RS,\ instance(p_1,\ p_2)) \Rightarrow (H_2,\ E,\ ES,\ p_3{\cdot}RS) \\ H_2(p_3) = p_4 \rightarrow p_5\end{array}}{(H_1,\ E,\ ES,\ RS,\ \textbf{con}\ (con,\ \overline{\tau}^{\,k})) \Rightarrow (H_2[l_1\ \mapsto\ con],\ E,\ ES,\ (l_1,\ p_5){\cdot}RS)} \tag{89}$$

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1){\cdot}RS_2) \\ E_2(con) = p_2 \qquad H_2(p_2) = p_3 \rightarrow p_4\end{array}}{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \textbf{con}\ (con,\ \Lambda)) \Rightarrow \\ \quad (H_2[l_2\ \mapsto\ con(l_1)],\ E_2,\ ES_2,\ (l_2,\ p_4){\cdot}RS_2)\end{array}} \tag{90}$$

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \qquad E_2(con) = p_2 \\ (H_2[p_3 \ \mapsto\ \overline{\tau}^{\,k}],\ E_2,\ ES_2,\ RS_2,\ instance(p_2,\ p_3)) \Rightarrow (H_3,\ E_2,\ ES_2,\ p_4\cdot RS_2) \\ H_3(p_4) = p_5 \to p_6\end{array}}{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{con}\ (con,\ \overline{\tau}^{\,k},\ \Lambda)) \Rightarrow \\ \quad (H_3[l_2 \ \mapsto\ con(l_1)],\ E_2,\ ES_2,\ (l_2,\ p_6)\cdot RS_2)\end{array}} \tag{91}$$

Comment: (Rules 88 to 91) Constructing a datatype value is analogous to applying the constructor function $\tau \to tn$ (or an instance of $\forall\ \overline{tv}^{\,k}.\ \tau \to tn$ for polymorphic constructors). Unary constructors require an argument Λ of type τ. A new constructor value is allocated on the value heap with associated type tn in the type heap.

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \\ E_2(con) = p_2 \qquad H_2(p_2) = p_3 \to p_4 \qquad H_2(l_1) = con(l_2)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{decon}\ (con,\ \Lambda)) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_2,\ p_3)\cdot RS_2)} \tag{92}$$

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \qquad E_2(con) = p_2 \\ (H_2[p_3 \ \mapsto\ \overline{\tau}^{\,k}],\ E_2,\ ES_2,\ RS_2,\ instance(p_2,\ p_3)) \Rightarrow (H_3,\ E_2,\ ES_2,\ p_4\cdot RS_2) \\ H_3(p_4) = p_5 \to p_6 \qquad H_3(l_1) = con(l_2)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{decon}\ (con,\ \overline{\tau}^{\,k},\ \Lambda)) \Rightarrow (H_3,\ E_2,\ ES_2,\ (l_2,\ p_5)\cdot RS_2)} \tag{93}$$

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda_1) \Rightarrow (H_2,\ E_2,\ ES_2,\ RS_2) \\ (H_2,\ E_2,\ ES_2,\ RS_2,\ \Lambda_2) \Rightarrow (H_3,\ E_3,\ ES_3,\ (l_2,\ p_2)\cdot(l_1,\ p_1)\cdot RS_3)\end{array}}{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{assign}\ (\Lambda_1,\ \Lambda_2)) \Rightarrow \\ \quad (H_3[l_1 \ \overset{upd}{\mapsto}\ \mathrm{c\_ref}(l_2)],\ E_3,\ ES_3,\ (l_{unit},\ p_{unit})\cdot RS_3)\end{array}} \tag{94}$$

Comment: (Rule 94) Assignment uses the update operation $l_1 \overset{upd}{\mapsto} \mathrm{c\_ref}(l_2)$ to update the reference at l_1 to $\mathrm{c\_ref}(l_2)$

A.5 Structured Expressions

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda_1) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \\ cmap = \{c^1 \ \mapsto\ \Lambda_2^1,\ \ldots,\ c^k \ \mapsto\ \Lambda_2^k\} \qquad H_2(l_1) = val \\ \Lambda_4 = \text{if } val \in \text{Dom } cmap \text{ then } cmap(val) \text{ else } \Lambda_3 \\ (H_2,\ E_2,\ ES_2,\ RS_2,\ \Lambda_4) \Rightarrow (H_3,\ E_3,\ ES_3,\ RS_3)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{switch}\ \Lambda_1\ \mathbf{case}\ (cmap,\ \Lambda_3)) \Rightarrow (H_3,\ E_3,\ ES_3,\ RS_3)} \tag{95}$$

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda^1) \Rightarrow (H_2,\ E_2,\ ES_2,\ RS_2)\ \cdots \\ \quad (H_k,\ E_k,\ ES_k,\ RS_k,\ \Lambda^k) \Rightarrow \\ \qquad (H_{k+1},\ E_{k+1},\ ES_{k+1},\ (l_k,\ p_k)\cdots(l_1,\ p_1)\cdot RS_{k+1})\end{array}}{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{tuple}\ \overline{\Lambda}^{\,k}) \Rightarrow \\ \quad (H_{k+1}[l_{k+1} \ \mapsto\ (l_1,\ \ldots,\ l_k)][p_{k+1} \ \mapsto\ (p_1,\ \ldots,\ p_k)],\ E_{k+1}, \\ \qquad ES_{k+1},\ (l_{k+1},\ p_{k+1})\cdot RS_{k+1})\end{array}} \tag{96}$$

Comment: (Rule 96) A tuple is constructed by evaluating its members $\overline{\Lambda}^k$ in left-to-right order. The resulting $(l,\ p)$ pairs are kept on the result stack RS until the last one is evaluated. A tuple $\overline{l}^k$ is then allocated on the value heap (with a corresponding type on the type heap) to hold the results.

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \\ H_2(l_1) = \overline{l_2}^k \qquad H_2(p_1) = \overline{p_2}^k\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{select}\ (i,\ \Lambda)) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_2^i,\ p_2^i)\cdot RS_2)} \tag{97}$$

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda_1) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \\ (H_2[p_2 \mapsto \sigma],\ E_2[lv \mapsto (l_1,\ p_2)],\ ES_2,\ E_2\cdot RS_2,\ \Lambda_2) \Rightarrow \\ \quad (H_3,\ E_3,\ ES_3,\ (l_3,\ p_3)\cdot E_4\cdot RS_3)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{let}\ (lv,\ \sigma) = \Lambda_1\ \mathbf{in}\ \Lambda_2) \Rightarrow (H_3,\ E_4,\ ES_3,\ (l_3,\ p_3)\cdot RS_3)} \tag{98}$$

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda_1) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \qquad H_2(l_1) = \overline{l_2}^k \\ (H_2[p_2^1 \mapsto \sigma^1,\ \ldots,\ p_2^k \mapsto \sigma^k],\ E_2[lv^1 \mapsto (l_2^1,\ p_2^1),\ \ldots,\ lv^k \mapsto (l_2^k,\ p_2^k)], \\ \quad ES_2,\ E_2\cdot RS_2,\ \Lambda_2) \Rightarrow (H_3,\ E_3,\ ES_3,\ (l_3,\ p_3)\cdot E_4\cdot RS_3)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS,\ \mathbf{let}\ \overline{(lv,\ \sigma)}^k = \Lambda_1\ \mathbf{in}\ \Lambda_2) \Rightarrow (H_3,\ E_4,\ ES_3,\ (l_3,\ p_3)\cdot RS_3)} \tag{99}$$

A.6 Function Expressions

$$\frac{\begin{array}{l}(H_1[l_1^1 \mapsto \Omega,\ \ldots,\ l_1^k \mapsto \Omega][p_1^1 \mapsto \sigma^1,\ \ldots,\ p_1^k \mapsto \sigma^k], \\ \quad E_1[v^1 \mapsto (l_1^1,\ p_1^1),\ \ldots,\ v^k \mapsto (l_1^k,\ p_1^k)],\ ES_1,\ RS_1,\ \Lambda_1^1) \Rightarrow \\ \qquad (H_2,\ E_2,\ ES_2,\ (l_2,\ p_2)\cdot RS_2) \\ (H_2[l_1^1 \overset{upd}{\mapsto} H_2(l_2)],\ E_2,\ ES_2,\ RS_2,\ \Lambda_1^2) \Rightarrow (H_3,\ E_3,\ ES_3,\ (l_3,\ p_3)\cdot RS_3)\ \cdots \\ \quad (H_k[l_1^{k-1} \overset{upd}{\mapsto} H_k(l_k)],\ E_k,\ ES_k,\ RS_k,\ \Lambda_1^k) \Rightarrow \\ \qquad (H_{k+1},\ E_{k+1},\ ES_{k+1},\ (l_{k+1},\ p_{k+1})\cdot RS_{k+1}) \\ (H_{k+1}[l_1^k \overset{upd}{\mapsto} H_{k+1}(l_{k+1})],\ E_{k+1},\ ES_{k+1},\ RS_{k+1},\ \Lambda_2) \Rightarrow \\ \quad (H_{k+2},\ E_{k+2},\ ES_{k+2},\ RS_{k+2})\end{array}}{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{fix}\ \overline{(lv,\ \sigma) = \Lambda_1}^k\ \mathbf{in}\ \Lambda_2) \Rightarrow \\ \quad (H_{k+2},\ E_{k+2},\ ES_{k+2},\ RS_{k+2})\end{array}} \tag{100}$$

Comment: (Rule 100) This rule achieves a simultaneous binding of a sequence of function closures (obtained from evaluating $\overline{\Lambda}^k$) to the variables $\overline{v}^k$. Initially, a dummy closure is allocated on the heap for each variable. The closure expressions are then evaluated in turn, and the dummy closures are updated to real closures. Thus, when the body expression Λ_2 is evaluated, all of the dummy closures will have been updated, and any closure which references another will do so correctly when evaluated.

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda_1) \Rightarrow (H_2,\ E_2,\ ES_2,\ RS_2)\\ (H_2,\ E_2,\ ES_2,\ RS_2,\ \Lambda_2) \Rightarrow (H_3,\ E_3,\ ES_3,\ (l_2,\ p_2)\cdot(l_1,\ p_1)\cdot RS_3)\\ H_3(l_1) = \langle\!\langle E_c,\ lv,\ \Lambda_c\rangle\!\rangle\\ (H_3,\ E_c[lv\ \mapsto\ (l_2,\ p_2)],\ ES_3,\ E_3\cdot RS_3,\ \Lambda_c) \Rightarrow\\ \quad (H_4,\ E_4,\ ES_4,\ (l_3,\ p_3)\cdot E_5\cdot RS_4)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \textbf{app}\ (\Lambda_1,\ \Lambda_2)) \Rightarrow (H_4,\ E_5,\ ES_4,\ (l_3,\ p_3)\cdot RS_4)} \tag{101}$$

Comment: (Rule 101) The function application rule applies the function expression Λ_1 (which evaluates to a closure) to the argument expression Λ_2. Firstly, both expressions are evaluated. The closure is then obtained from the result of Λ_1, and the result of Λ_2 is bound to the variable v in the closure environment E_1. The body of the closure Λ_3 is then evaluated in this environment. The previous environment E is then restored. The result of the function application remains on the result stack.

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda_1) \Rightarrow (H_2,\ E_2,\ ES_2,\ RS_2)\\ (H_2,\ E_2,\ ES_2,\ RS_2,\ \Lambda_2^1) \Rightarrow (H_3,\ E_3,\ ES_3,\ RS_3)\ \cdots\\ \quad (H_{k+1},\ E_{k+1},\ ES_{k+1},\ RS_{k+1},\ \Lambda_2^k) \Rightarrow\\ \qquad (H_{k+2},\ E_{k+2},\ ES_{k+2},\ (l_2^k,\ p_2^k)\cdot(l_2^1,\ p_2^1)\cdot(l_1,\ p_1)\cdot RS_{k+2})\\ H_{k+2}(l_1) = \langle\!\langle E_c,\ \overline{lv}^k,\ \Lambda_c\rangle\!\rangle\\ (H_{k+2},\ E_c[lv^1\ \mapsto\ (l_2^1,\ p_2^1),\ \ldots,\ lv^k\ \mapsto\ (l_2^k,\ p_2^k)],\\ \quad ES_{k+2}, E_{k+2}\cdot RS_{k+2},\ \Lambda_c) \Rightarrow (H_{k+3},\ E_{k+3},\ ES_{k+3},\ (l_3,\ p_3)\cdot E_{k+4}\cdot RS_{k+3})\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \textbf{app}\ (\Lambda_1,\ \overline{\Lambda_2}^k)) \Rightarrow (H_{k+3},\ E_{k+4},\ ES_{k+3},\ RS_{k+3})} \tag{102}$$

A.7 Exceptions

$$\frac{(H_1,\ E_1,\ (),\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ (),\ RS_2)}{(H_1,\ E_1,\ (),\ RS_1,\ \textbf{raise}\ \Lambda) \Rightarrow \textbf{halt}\ (H_2,\ E_2,\ (),\ RS_2)} \tag{103}$$

Comment: (Rule 103) If there are no closures on the exception stack then a raised exception will not be handled. The effect of an un-handled exception is to halt the evaluation of the abstract machine.

$$\frac{\begin{array}{l}(H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda) \Rightarrow (H_2,\ E_2,\ (l_1,\ p_1)\cdot ES_2,\ (l_2,\ p_2)\cdot RS_2)\\ H_2(l_1) = \langle\!\langle E_c,\ lv,\ \Lambda_c\rangle\!\rangle\\ (H_2,\ E_c[lv\ \mapsto\ (l_2,\ p_2)],\ ES_2,\ E_2\cdot RS_2,\ \Lambda_c) \Rightarrow\\ \quad (H_3,\ E_3,\ ES_3,\ (l_3,\ p_3)\cdot E_4\cdot RS_3)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \textbf{raise}\ \Lambda) \Rightarrow (H_3,\ E_4,\ ES_3,\ (l_3,\ p_3)\cdot RS_3)} \tag{104}$$

Comment: (Rule 104) If an exception is raised, and the exception stack is non-empty, the closure at the top of the exception stack is evaluated (see Rule 101).

$$\frac{\begin{array}{l}\Lambda_2 = (\mathbf{fn}\ (lv,\ \tau_1 \to \tau_2) = \Lambda_3) \\ (H_1,\ E_1,\ ES_1,\ RS_1,\ \Lambda_2) \Rightarrow (H_2,\ E_2,\ ES_2,\ (l_1,\ p_1)\cdot RS_2) \\ (H_2,\ E_2,\ (l_1,\ p_1)\cdot ES_2,\ RS_2,\ \Lambda_1) \Rightarrow (H_3,\ E_3,\ (l_2,\ p_2)\cdot ES_3,\ RS_3)\end{array}}{(H_1,\ E_1,\ ES_1,\ RS_1,\ \mathbf{handle}\ \Lambda_1\ \mathbf{with}\ \Lambda_2) \Rightarrow (H_3,\ E_3,\ ES_3,\ RS_3)} \tag{105}$$

Comment: (Rule 105) This rule ensures that an exception raised in Λ_1 is handled by Λ_2 (which is syntactically a closure, as ensured by the equation $\Lambda_2 = (\mathbf{fn}\ (lv,\ \tau_1 \to \tau_2) = \Lambda_3)$). This amounts to simply applying Rule 86 to Λ_2 and placing it on the exception stack while Λ_1 is evaluated. The **raise** rule performs the actual evaluation of the exception handler.

References

[1] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised 1997.* The MIT Press, 1997.

[2] David C.J. Matthews and Thierry Le Sergent. LEMMA Interface Definition. Technical Report ECS-LFCS-95-316, LFCS, Division of Informatics, University of Edinburgh, January 1995.

[3] David C.J. Matthew. A Distributed Concurrent Implementation of Standard ML. In *Proceedings of EurOpen Autumn 1991 Conference*, September 1991. Also published as LFCS Technical Report ECS-LFCS-91-17.

[4] Greg Morrisett and Robert Harper. Semantics of Memory Management for Polymorphic Languages. Technical Report CMU-CS-96-176, School of Computer Science, Carnegie Mellon University, September 1996. Also published as Fox Memorandum CMU-CS-FOX-96-04.

[5] Chris Walton, Dilsun Kırlı, and Stephen Gilmore. An Abstract Machine for Module Replacement. In Stephan Diehl and Peter Sestoft, editors, *Proceedings of the Workshop on Principles of Abstract Machines*, pages 73–87, September 1998. Also published as Technical Report A 02/98 Universität Des Saarlandes.

[6] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42. Springer-Verlag, September 1992.

[7] M. Tofte and J. Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[8] Andrew Tolmach. Tag-free Garbage Collection using Explicit Type Parameters. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 1–11. ACM Press, June 1994.

[9] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed Closure Conversion. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, January 1996.

[10] Andrew W. Appel. *Compiling with Continuations*, chapter 12. Cambridge University Press, 1992.

[11] Christopher D. Walton and Bruce J. McAdam. The C-LEMMA Memory Interface on the Cray T3D. Technical Report ECS-LFCS-97-362, LFCS, Division of Informatics, University of Edinburgh, July 1997.

[12] David C.J. Matthews and Thierry Le Sergent. LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection. Technical Report ECS-LFCS-95-325, LFCS, Division of Informatics, University of Edinburgh, June 1995.

[13] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, pages 52–60, August 1991.

[14] Thierry Le Sergent and David C J Matthews. Adaptive selection of protocols for strict coherency in distributed shared memory. Technical Report ECS-LFCS-94-306, LFCS, Division of Informatics, University of Edinburgh, September 1994.

[15] Saleh E. Abdullahi and Graem A. Ringwood. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Computing Surveys*, 30(3):330–373, September 1998.

[16] Dave Berry. *Generating Program Animators from Programming Language Semantics*. PhD thesis, LFCS, Division of Informatics, University of Edinburgh, June 1991. Thesis No. CST-79-91.

[17] Xavier Leroy. *Polymorphic Typing of an Algorithmic Language*. PhD thesis, INRIA, 1992. Thesis No. 1778.

[18] Kevin Mitchell. Concurrency in a Natural Semantics. Technical Report ECS-LFCS-94-311, LFCS, Division of Informatics, University of Edinburgh, December 1994.