Future Generation Computer Systems 16 (2000) 793–808

An abstract machine model of dynamic module replacement

Chris Walton, Dilsun Kırlı, Stephen Gilmore[*]

Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, Scotland, UK

Accepted 6 May 1999

Abstract

In this paper we define an abstract machine model for the $m\lambda$ typed intermediate language. This abstract machine is used to give a formal description of the operation of run-time module replacement for the programming language Dynamic ML. The essential technical device which we employ for module replacement is a modification of two-space copying garbage collection. We show how the operation of module replacement could be applied to other garbage-collected languages such as Java. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Standard ML; Copying garbage collection; Typed abstract machine; Dynamic module replacement; Java

1. Introduction

We have previously presented the high-level design of Dynamic ML, a variant of the Standard ML programming language which incorporates a facility for the replacement of modular components during program execution [1]. This useful facility builds upon existing compiler technology which permits the separate compilation of modular units of a Standard ML program. A suitable application for Dynamic ML would be the implementation of a distributed system where it is necessary to correct errors, improve run-time performance or reduce memory use, all without interrupting the execution of the system. We use a modified form of memory management to implement this idea.

The definition of Standard ML [2] is a formal description of the language which acts as a solid scientific platform where experiments in programming language design may be conducted. Any alteration to the Standard ML language such as ours should be investigated in the terms of the definition. However, as readers of the definition will know, it is silent on the topic of memory management except to say that "there are no (semantic) rules concerning disposal of inaccessible addresses" ([2], p. 42). The definition also separates the static and the dynamic semantics in such a way that the typing information inferred at compile-time is discarded before run-time. However, Dynamic ML needs some type information at run-time. These differences have motivated our work on a novel semantic model that would form a suitable setting for the formal definition of Dynamic ML. That model is presented in this paper.

Other authors have argued for the usefulness of a semantic model of memory management in making precise implementation notions such as memory leaks and tail recursion optimisation, developing suitable abstract machine

[*] Corresponding author.
E-mail address: stg@dcs.ed.ac.uk (S. Gilmore)

models of memory management for this purpose [3]. Our abstract machine model for Dynamic ML serves a different purpose and this has led to the creation of a significantly different abstract machine than those used by previous authors. An essential feature of our machine is the modelling of user program exceptions, which other authors do not include.

In Section 2 we present the main idea of our module replacement model by means of an example. Sections 3 and 4 introduce respectively the language on which our discussion is based and the abstract machine which is used in specifying the dynamic semantics of this language. In Section 5 we give the formal definition of the garbage collection with replacement operation. Section 6 describes how the main idea of our work can be applied to other garbage-collected languages such as Java. Section 7 discusses the practicality of our model and the related work of others. We conclude in Section 8.

2. A model for module replacement

We introduce our first-order module-level replacement with an example to give the reader an informal understanding of its use in practice. Standard ML has interfaces called *signatures* and modules called *structures*. In our replacement model we allow the replacement of signatures by other signatures and structures by other structures, under reasonably generous conditions [1]. As our running example we consider the replacement of one implementation of a name table with another which is functionally equivalent but offers improved performance. Both implementations match the TABLE signature shown below.

```
signature TABLE =
sig
  type table
  type ame = string
  val empty: table
  val insert: name × table → table
  val member: name × table → bool
end;
```

We provide a facility for expressing such a replacement which ensures that the data values already present in memory cannot be used in ways which are inconsistent with their type. The replacement operation is expressed by allowing

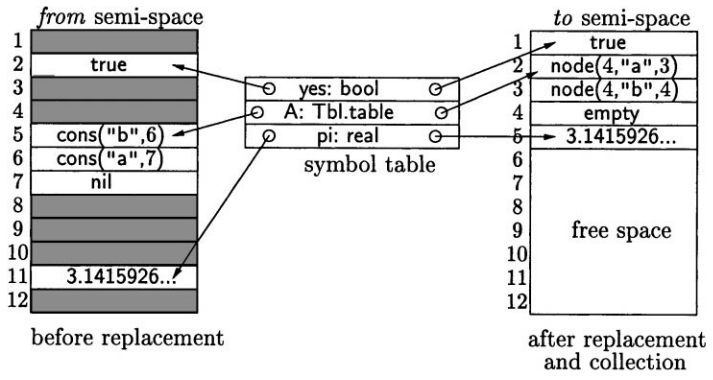


Fig. 1. Code replacement with type update.

the user to abstract over a Tbl structure which is specialised to implement a name table as a list of character strings. The Standard ML terminology for a structure abstraction is a *functor*. The functor body describes a structure which implements name tables as binary search trees. The user-defined datatype for tables declares that a table can either be empty or it can be constructed from a name as its root and left and right sub-trees which are name tables themselves. The functions for inserting a name into a table and testing for membership use pattern-matching to de-construct the name tables and bind values to the formal parameters (named s, l, v and r for the string to be added, the left sub-tree, the value at the root and the right sub-tree, respectively). The functor also contains functions to convert from the old representations to the new. These functions are placed inside an Install structure. They follow a convention of using the same identifier as the type which they update. The first conversion function is the identity function and the second one is an application of the Standard ML Basis library function implementing folding a function across a list with right associativity. This method of structure replacement is encoded as a Dynamic ML functor below.

```
functor InstallTable (Tbl: TABLE where type table = string list) :> TABLE =
struct

  type name = string
  datatype table = empty | node of table × name × table

  fun insert (s, empty) = node (empty, s, empty)
    | insert (s, node (l, v, r)) =
        if s < v then node (insert (s, l), v, r)
        else if s > v then node (l, v, insert (s, r)) else node (l, v, r)
  fun member (s, empty) = false
    | member (s, node (l, v, r)) =
        if s < v then member (s, l)
        else if s > v then member (s, r) else true

  structure Install = struct
    val name: Tbl.name → name = fn x ⇒ x
    val table: Tbl.table → table = List.foldr insert empty
  end

end;
```

Through the use of the InstallTable functor, a Dynamic ML programmer could replace a structure which implemented tables as (either sorted or unsorted) lists with one which implemented them as binary search trees. This is an example of a very simple modification which would improve the performance of the insert and member operations. However, more sophisticated improvements would also be made by the same method: defining a functor which maps the old implementation to the new one and provides functions to convert from the old types to the new. In both cases, it is critical that the types under replacement are abstract ones (with only the type identifier given in the signature) in order that functions outside the structure were not able to depend on a particular choice of concrete representation for a type, thereby preventing its replacement.

We propose to perform the code replacement operation during garbage collection. A functor, such as the one shown above, is compiled separately. We then invoke the garbage collection operation extended with the application of the replacement functions from the Install structure to any values of the type under replacement. After completion of the copying with replacement, it is possible to dispose of the outdated version of the structure under modification (in the *from* semi-space), and switch to use the new version (in the *to* semi-space) which now contains the data values of the newly introduced replacement types. This is illustrated in Fig. 1 where a list representation of a name table containing the names *b* and *a* is replaced by the corresponding tree representation. Values of types not under replacement are unaltered: this includes values of built-in types such as booleans and real numbers.

The functions which are executed during code replacement are unrestricted Standard ML functions which may diverge upon application or raise an exception to signal an inability to continue processing. Our method of recovery is to rollback the garbage collection operation when any exception is raised. We revert to using the *from* semi-space of data values and the old types and we continue with the execution of the old program code.

One remaining issue that must be addressed is the replacement of code that is active. It is important to take steps to prevent the replacement of any function which is currently being executed. If allowed, such unguarded replacements would result in undefined behaviour as the continuation point would no longer exist. One way of preventing unguarded replacements is to maintain the old version of the function (and its data) until the present invocation of the function has terminated. However, this method of prevention significantly complicates the replacement operation. Consequently, our solution only permits the replacement of functions that are inactive. This is not a severe restriction because in the language on which replacement is defined, iteration is performed by recursion and replacement may be performed between recursive function calls.

3. The $m\lambda$ language

In order to formalise the replacement operation described in the previous section we first define a call-by-value lambda language $m\lambda$. This language is representative of a typical typed intermediate language used in the current state-of-the-art Standard ML compilers [4–7]. By basing replacement on such an intermediate language, we obtain an operation that is applicable to the whole of Standard ML, yet avoid a great deal of the complexity. For example, pattern matching is converted into switch statements by the higher-level match compiler. Furthermore, we can assume that the $m\lambda$ program is well-typed. For brevity, we have restricted our attention here to a purely-functional monomorphic lambda language. However, we note that including polymorphism at the Dynamic ML language level does not change the resulting replacement operation because a polymorphic source language may be compiled down into a monomorphic typed intermediate language. Adding references also does not change the resulting replacement operation, as the machine state is stored on the heap. Thus, references may simply be treated as values of datatypes.

The syntax of the $m\lambda$ language is given in Fig. 2. The syntactic categories of the language include special constants of types unit, integer, real, and string; value constructors such as c_true; exception constructors such as e_match; type

Types	τ	::=	$tn \mid tn(\tau) \mid \{\overline{\tau}^k\} \mid \tau_1 \to \tau_2$
Program	P	::=	$(\overline{D}^k,\ X)$
Datatype	D	::=	**datatype** tn **of** $\overline{(con,\ \tau)}^k$
Expression	X	::=	**scon** $scon$
		|	**con** con | **con** $(con,\ X)$
		|	**decon** $(con,\ X)$
		|	**excon** $excon$ | **excon** $(excon,\ X)$
		|	**dexcon** $(excon,\ X)$
		|	**record** $\overline{X}^k$
		|	**select** $(i,\ X)$
		|	**var** v
		|	**let** $v = X_1$ **in** X_2
		|	**fix** $\overline{(v,\ \tau) = X_1}^k$ **in** X_2
		|	**fn** $(v,\ \tau_1 \to \tau_2) = X$
		|	**app** $(X_1,\ X_2)$
		|	**switch** X_1 **case** $(c \overset{map}{\mapsto} X_2,\ X_3)$
		|	**exception** $(excon,\ \tau)$ **in** X
		|	**raise** X
		|	**handle** X_1 **with** X_2

Fig. 2. Syntax of the $m\lambda$ language.

names such as t_bool. Variables are bound uniquely to values generated by the evaluation of expressions. The types are constructor types (which may be either nullary or unary), record types, and function types. Constructor types include the basic types, as required by the special constants; value constructor types; and exception constructor types. A program consists of a sequence of datatype declarations followed by a single expression. A datatype declaration consists of a unique type name and a sequence of typed constructors. The expressions divide into those for constructing and de-constructing values, defining and manipulating variables, and controlling the order of evaluation.

Notation: We use the meta-variables *scon* for special constants, *con* and *excon* for value and exception constructors with c ranging over all three of these and i over special constants of integer type. We use *tn* for type names and v for variables. We use p for type pointers and l for value locations. A set is defined by enumerating its members in braces, such as $\bar{\bar{x}} = \{a, b, c, d\}$ with $\emptyset$ for the empty set. A sequence is an ordered list of members of a set, e.g. $\bar{x}^k = (a, b, c, a)$. The ith element of a non-empty sequence is written x^i, where $0 < i \leq k$. A finite map from $\bar{x}^k$ to $\bar{y}^k$ is defined: $x \overset{\text{map}}{\mapsto} y = \{x^1 \mapsto y^1, \ldots, x^k \mapsto y^k\}$ (the elements of $\bar{x}^k$ must be unique). The domain Dom and range Rng are the sets of elements of $\bar{x}^k$ and $\bar{y}^k$, 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.

4. The $m\lambda$ abstract machine

The dynamic semantics of $m\lambda$ is formalised in this section by a transition relation between states of an abstract machine. The organisation of our abstract machine has some features in common with the $\lambda_{\text{gc}}^{\rightarrow\forall}$ abstract machine [3] which is used in the formal description of the behaviour of the TIL/ML compiler. However, the resulting transitions differ considerably as $m\lambda$ is significantly different from $\lambda_{\text{gc}}^{\rightarrow\forall}$. One important way in which it differs is that $m\lambda$ does not adopt the named-form representation of expressions and types.

The syntax of the abstract machine is given in Fig. 3. 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 objects of the program, while the environment provides a view of the heap relevant to the fragment of the program being evaluated (for example, 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.

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 $m\lambda$ language, and the heap values belong to the heap types. Nullary constructors *scon*, *con*, and *excon* all have type *tn*. Unary constructors $con(l)$ and $excon(l)$ have type $tn(p)$. Records $\{\bar{l}^k\}$ have type $\{\bar{p}^k\}$, and closures $\langle\langle E, v, X\rangle\rangle$ have type $p_1 \rightarrow p_2$. The type heap and value heap are represented by finite maps, as locations and type-pointers may be bound only once. It is important to note that we can only determine the shape of the data at a particular location by examining its corresponding type. Thus, each heap location will be paired with a type-pointer: (l, p). This is essential for implementing tag-free garbage collection in the following section.

The following syntactic conventions are used for allocating heap objects: the term $H[l_1 \mapsto val_1, \ldots, l_k \mapsto val_k]$ allocates values $val_1, \ldots, val_k$ on the value heap, binding them to fresh locations $l_1, \ldots, l_k$. Correspondingly, the term $H[p_1 \mapsto \tau_1, \ldots, p_k \mapsto \tau_k]$ allocates types $\tau_1, \ldots, \tau_k$ on the type heap, binding them to fresh pointers $p_1, \ldots, p_k$. There are no corresponding operations for removing objects from the heap as this is achieved through garbage collection. However, the implementation of the fixed-point expression which is used to implement recursive functions requires a heap-update operation. As a special case, $H[l \mapsto \Omega]$ allocates a dummy closure on the value heap bound to a fresh location l. This location can subsequently be updated with a mapping to a new closure.

The environment records the allocation of $m\lambda$ values, mapping them to heap locations and type-pointers. As the identifiers and variables have been made unique, via the compilation into $m\lambda$, their environments are represented by

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

Fig. 3. Syntax of the $m\lambda$ abstract machine.

finite maps, with the exception of the type environment where it is sufficient just to use a set. 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 type-pointer p in the constructor environment. Similarly, $E[excon \mapsto p]$ and $E[v \mapsto (l,\ p)]$ denote the binding of exception constructors and variables respectively to heap locations and type-pointers in the environment. There are no operations for removing objects 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, objects 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 $(H,\ E,\ ES,\ RS,\ P) \Rightarrow (H',\ E',\ ES',\ RS')$, where P is an $m\lambda$ program, $(H,\ E,\ ES,\ RS)$ is the initial machine state (as illustrated in Fig. 4), and

M	=	$(H,\ E,\ ES,\ RS)$
H	=	$(TH,\ VH)$
TH	=	$\{p_1 \mapsto \text{t\_unit} \to \text{t\_bool},\ p_2 \mapsto \text{t\_unit} \to \text{t\_exn}\}$
VH	=	$\emptyset$
E	=	$(TE,\ CE,\ EE,\ VE)$
TE	=	$\{\text{t\_unit},\ \text{t\_int},\ \text{t\_real},\ \text{t\_string},\ \text{t\_bool},\ \text{t\_exn}\}$
CE	=	$\{\text{c\_true} \mapsto p_1,\ \text{c\_false} \mapsto p_1\}$
EE	=	$\{\text{e\_match} \mapsto p_2,\ \text{e\_bind} \mapsto p_2\}$
VE	=	$\emptyset$
ES	=	$()$
RS	=	$()$

Fig. 4. Initial machine state.

(H', E', ES', RS') is the final machine state. This top-level transition decomposes into a sequence of transitions of the form $(H, E, ES, RS, D) \Rightarrow (H', E', ES', RS')$ for processing the datatypes D, followed by a sequence $(H, E, ES, RS, X) \Rightarrow (H', E', ES', RS')$ for evaluating the expression X.

There are three possible outcomes which can result from the evaluation of this expression. These three outcomes are well-known to programmers working within the simpler Standard ML programming discipline: termination, exceptional termination and non-termination. Firstly, the sequence 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. Thirdly, the machine may encounter an infinite sequence of transitions and fail to terminate.

5. Garbage collection with replacement

In Section 2 we have explained how we extend the traditional two-space copying garbage collection to implement our replacement operation. In this section, we give the formal definition of this extended garbage collection as it is used in the abstract machine defined in Section 4. The replacement operation has been presented in terms of the use of the modular constructs of Standard ML. However, for brevity we restrict our discussion here to the simpler non-modular language presented in Section 3.

We will consider the case where we have the information represented by a semantic object defined as follows: $RM ::= p_{\mathrm{old}} \overset{\mathrm{map}}{\mapsto} (l_{\mathrm{rep}}, p_{\mathrm{rep}})$. The domain of the replacement map Dom RM is the set of the pointers to the types that are to be dynamically replaced. Each element p_{old} of the domain is mapped to a location/type-pointer pair $(l_{\mathrm{rep}}, p_{\mathrm{rep}})$. The location contains the closure of the function which is to execute the replacement operation and the type-pointer points to the type which is to replace the old type.

In Dynamic ML this information is extracted from the result of the evaluation of the sub-structure Install which contains the user defined functions for the replacement operation. The replacement map thus obtained would be $\{p_{\mathrm{Tbl.name}} \mapsto (l_{\mathrm{name}}, p_{\mathrm{name}}), p_{\mathrm{Tbl.table}} \mapsto (l_{\mathrm{table}}, p_{\mathrm{table}})\}$. We define garbage as the objects that are not reachable either directly or indirectly from the environment, exception stack, or result stack. Garbage collection may take place before or after any transition of the $m\lambda$ abstract machine dropping the bindings of the unreachable objects provided that this does not change the observable behaviour of the program.

Garbage collection is defined as a rewriting system between configurations of the form (S, RM, H_f, H_t). The replacement map denoted by RM is the auxiliary data structure which provides the information necessary for the replacement operation; the traditional two-space copying garbage collection corresponds to the case where RM is empty. Initially, the scan stack S contains all of the pointers p and (l, p) pairs in E, ES, and RS. Following the rules listed in Appendix B, heap objects are copied from the semi-space H_f to the semi-space H_t until the scan stack is empty. We can incorporate the garbage collection operation in the dynamic semantics of our language explicitly by means of the following evaluation rule:

$$\frac{\begin{array}{l}(ES \cdot RS \cdot FE(E), RM, H_f, \emptyset) \Rightarrow^*_{\mathrm{gc}} ((), \emptyset, H_f, H_t) \\ (H_t, E, ES, RS, X) \Rightarrow (H_1, E_1, ES_1, RS_1)\end{array}}{(H_f, E, ES, RS, X) \Rightarrow (H_1, E_1, ES_1, RS_1)}$$

where $\Rightarrow^*_{\mathrm{gc}}$ stands for the repeated application of the $\Rightarrow_{\mathrm{gc}}$ rules.

The informal understanding of the $\Rightarrow_{\mathrm{gc}}$ rules is as follows:

Rule R0 is applied when the scan stack is empty. This signals the end of the garbage collection operation. The replacement map is discarded in order for subsequent garbage collections to operate correctly.

Rules R1 and R1a and R1b are applied when the top of the scan stack is a pair of a location and a type-pointer (l, p) and the value in the location has not yet been copied to the H_t semi-space, i.e. $l \notin$ Dom H_t.

$$FE(E) = \text{Rng } CE{\cdot}\text{Rng } EE{\cdot}\text{Rng } VE$$

$$\begin{aligned}
FP(tn) &= () \\
FP(tn(p)) &= (p) \\
FP(\{\overline{p}^{k}\}) &= (p^{1}\cdots p^{k}) \\
FP(p_1 \rightarrow p_2) &= (p_1{\cdot}p_2)
\end{aligned}$$

$$\begin{aligned}
FL(H,\ l,\ tn) &= () \\
FL(H,\ l_1,\ tn(p)) &= (l_2,\ p) \quad \text{where } tn = \text{t\_exn and } H(l_1) = excon(l_2) \\
FL(H,\ l_1,\ tn(p)) &= (l_2,\ p) \quad \text{where } tn \neq \text{t\_exn and } H(l_1) = con(l_2) \\
FL(H,\ l,\ \{\overline{p}^{k}\}) &= (l^{1},\ p^{1})\cdots(l^{k},\ p^{k}) \quad \text{where } H(l) = \{\overline{l}^{k}\} \\
FL(H,\ l,\ p_1 \rightarrow p_2) &= FE(E) \quad \text{where } H(l) = \langle\!\langle E,\ v,\ X\rangle\!\rangle
\end{aligned}$$

Fig. 5. Auxiliary functions for garbage collection.

In R1 the type of the value reveals that it need not be replaced. As a result, the value in the H_f semi-space is copied to the H_t semi-space. The free locations and the type pointers of the allocated value are added to the scan stack.

R1a and R1b are variants of R1 where the type of the value indicates that the value is to be replaced, i.e. $p \in \text{Dom}\, RM$. Consecutive lookups in the replacement map and the heap yield the closure of the replacement function that is to be applied to the value currently being scanned. The code of the closure is evaluated in the environment extended by the binding of the scanned value. Note also that the disjoint union of the two semi-spaces is assumed as the heap because the code may be referring to some location or type-pointer that has already been copied.

There are two possible outcomes for the garbage collection operation. Either evaluation ends successfully or an exception is raised by one of the functions which is updating the values from the old type to the new one. These two cases are distinguished by inspecting the type of the most recent result which is at the top of the result stack. The first case is captured by R1a. The new value is copied to the H_t semi-space and the scan stack is arranged as in R1. The second case is captured by R1b where the top of the stack indicates that a top level exception has been raised. According to our implementation model we rollback the garbage collection operation and revert to using the H_f semi-space values. This is indicated by setting the scan stack to empty and identifying H_t with H_f. The replacement map is discarded as in R0.

R2 is applied when the top of the scan stack is a location/type-pointer pair and the value in the location has already been copied to the H_t semi-space. It simply skips this location and continues with the rest of the scan stack. R4 is exactly like R2 but skips over a type-pointer instead of a location.

R3 and R3a are applied when the top of the scan stack is a type-pointer and the type-pointer has not yet been copied to the H_t semi-space. R3 deals with the case where the type need not be replaced. The free pointers of the allocated type are added to the scan stack and the old representation of the type is copied to the H_t semi-space. R3a deals with the case where the old representation of the type is to be replaced by the new representation.

The functions FE, FP and FL employed in the rewriting rules compute the free location/type-pointer pairs $(l,\ p)$ and type-pointers p. These rules are given in Fig. 5.

6. Replacement in Java

Our discussion of module replacement has been exclusively framed in the context of Standard ML. However, given an appropriate abstract machine and notion of replacement, the operation could be applied to a number of different garbage-collected languages. Indeed, such a re-use of the idea is appealing because useful, soundly-engineered products such as Java can be deployed in contexts where it is presently difficult to apply Standard ML. These would include embedded systems and also application domains where interoperability is an important consideration. Thus,

Machine State	M	::=	$(H,\ E,\ ES,\ RS)$
Heap	H	::=	$(TH,\ VH)$
Type Heap	TH	::=	$p \overset{map}{\mapsto} ty$
Heap Types	ty	::=	$pt \mid cn \mid p[\,] \mid \overline{p_1}^{k} \to p_2 \mid \texttt{null}$
Value Heap	VH	::=	$l \overset{map}{\mapsto} obj$
Heap Objects	obj	::=	$c \mid \langle v \overset{map}{\mapsto} l,\ m \overset{map}{\mapsto} (l,\ p)\rangle \mid [\bar{l}^{k}] \mid$ $\langle\!\langle \overline{v}^{j},\ \overline{S}^{k} \rangle\!\rangle \mid null$
Environment	E	::=	$(TE,\ CE,\ IE,\ VE)$
This	TE	::=	$this \mapsto (l,\ p)$
Class Env.	CE	::=	$cn \overset{map}{\mapsto} (cn',\ \bar{\bar{i}},\ FE,\ ME)$
Interface Env.	IE	::=	$i \overset{map}{\mapsto} (i',\ ME)$
Field Env.	FE	::=	$v \overset{map}{\mapsto} p$
Method Env.	ME	::=	$m \overset{map}{\mapsto} (l,\ p)$
Variable Env.	VE	::=	$v \overset{map}{\mapsto} (l,\ p)$
Exception Stack	ES	::=	$() \mid (l,\ p)\cdot ES$
Result Stack	RS	::=	$() \mid (l,\ p)\cdot RS$

Fig. 6. Syntax of the JAM.

we are currently considering the dynamic replacement of loaded classes in the Java run-time system. As it stands, the Java Virtual Machine (JVM) is not suitable as it does not retain enough typing information. However, we have designed a similar abstract machine for a subset of Java that is compatible with our replacement model.

The syntax of the Java Abstract Machine (JAM) appears in Fig. 6. The organisation of the machine is identical to the $m\lambda$ machine, but the contents of the heap and the environment have been adapted for the Java language. As before, the heap types correspond directly to types in the Java language, and the heap objects belong to the heap types. The heap types *ty* are primitive types *pt*, class instance types *cn*, array types *p*[ ], method types $\bar{p}_1^k \to p_2$, and the null type null. The corresponding heap objects *obj* are constants *c*, class objects $\langle v \overset{\text{map}}{\mapsto} l,\ m \overset{\text{map}}{\mapsto} (l,\ p)\rangle$, array objects $[\bar{l}^k]$, method (and constructor) closures $\langle\langle \bar{v}^j,\ \bar{S}^k \rangle\rangle$, and the *null* object.

Notation: The meta-variables cn, i, m, v, and c are used for class names, interface names, method names, variables, and constants respectively. The meta-variable *pt* denotes the primitive types (`int`, `float`, `boolean`, etc.).

The environment contains the class hierarchy and maps fields, methods, and variables to objects on the heap. When evaluating inside a class body, *this* is mapped to the current class object. The hierarchy is described by the class environment *CE* which maps a class name *cn* to its superclass *cn*′. The top of the hierarchy is assumed to be a class named `Object` which has itself as superclass.

The Java Abstract Machine presented above would provide the basis for a rigorous description of another application of dynamic code replacement. However, the completion of such a description remains as further work. We now turn to the consideration of practical application of the technique of dynamic code replacement.

7. Implementation issues

Users of state-of-the-art compilers for modern programming languages have become accustomed to complex program analyses. These can safely deliver impressive performance benefits in terms of run-time and memory usage. Modern programming languages also offer access to a more sophisticated model of computation incorporating advanced features such as remote evaluation or code mobility. In this setting it is all too easy to invent a new paradigm for program execution and to claim that it can be implemented efficiently because modern compilers and run-time systems offer so much functionality and convenience. In this section we would like to provide a more concrete explanation of the key implementation technology which could be used to provide an efficient implementation of the code replacement operation which we have described. Furthermore, experimental systems already exist which make use of dynamic code replacement for related purposes such as improving data locality and we include a brief description of these here.

Languages in the Standard ML family are strongly typed. In order to enforce the application of the type-checking stage these language make a strict distinction between *elaboration* and *evaluation*, insisting that programs which have not successfully elaborated cannot be evaluated at all. The rigid ordering of these two stages prohibits the execution of any programs which attempt to use data values in ways which are not allowed by their type and thus eliminates a large number of software errors which would manifest themselves at run-time if working in an untyped programming language. However, several authors have observed that two stages are not enough for complex applications such as program generators. This has led to approaches such as the *multi-stage* programming paradigm for MetaML [9], *staged type inference* [10] and the *staged compilation* paradigm for the language Modal ML [11]. The last of these is the most closely related to our own approach because it has demonstrated the effectiveness of the use of run-time code generation by Lee and colleagues in the development of the Fabius compiler for ML [12]. Using this technology it is possible for us to eliminate the run-time penalties incurred by the use of abstract types in module specifications by exploiting the underlying representation of an abstract type and re-compiling at run-time when the replacement module is available. Further, many other benefits come from the use of run-time code generation including those associated with partial evaluation since it is possible to take advantage of values which are not known until run-time. Other standard compiler optimisations such as elimination of array-bounds checking and loop unrolling also become more profitable in this setting.

Other researchers are considering similar ideas in the setting of dynamic object-oriented languages such as Java. Java differs from ML-like languages in several ways. For example, it does not provide the same level of abstraction provided by Standard ML signatures. Java interfaces allow us to abstract method declarations, but not types. Without this abstraction, replacing one Java class with another will force the rebuilding of all of its subclasses and all classes that reference it. Andersson et al. [8] have proposed one solution: they perform replacement of objects of the outdated class as they are accessed, meaning that both versions of the class are active at the same time. Replaced objects are garbage-collected as the computation proceeds, and whenever all of the objects of the old version of the class have been replaced the class object will have no more references and it can then be garbage-collected also.

A number of other researchers have investigated the use of garbage collection to improve the run-time behaviour of programs. For example, [13] outlines a dynamic profiling algorithm for exploring the data access patterns of a program in the Cecil language. During garbage collection, this information is used to regroup the data to improve locality. A similar technique is also used in the Java Hotspot Virtual Machine [14]. The authors of [15] have implemented a general-purpose run-time system combining dynamic profiling and optimisation, as described above, with dynamic replacement of code and data. The intention of this system is to allow the optimisation of the code by replacing procedures with more efficient ones during execution. The technique used is very similar to the one presented in this paper. Replacement is performed by building a list of translation tuples $(ptr_{\mathrm{old}}, ptr_{\mathrm{new}})$ analogous to our replacement map. During garbage collection, all occurrences of ptr_{old} are replaced by ptr_{new}. However, without the type preservation, installation and rollback mechanisms which we have presented in this paper, there is no guarantee that the notion of replacement from [15] is valid.

8. Conclusions

Modern compilers for higher-order typed programming languages use typed intermediate languages to structure the compilation process. We have provided an abstract machine definition of a small functional language which is representative of these. This has allowed us to define precisely the operation of dynamic module replacement which is used in Dynamic ML.

In composing the definition of Standard ML, the authors chose not to give an account of the operation of garbage collection, which most compilers for that language provide. This was the right decision when focusing upon the abstract description of a sophisticated high-level language such as Standard ML. Our concern was to describe part of the operation of an executing computation, with access to values described by concrete manifestations.

The use of an abstract machine notation has allowed us to isolate the novel feature of interest from our language. We have been able to present its definition separately from other aspects such as syntax and type-correctness. For our purposes, the use of an abstract machine has established the right level of detail. In addition, it provides an implementor with an unambiguous and precise description of the operation of module-level code replacement.

Acknowledgements

The comments from the anonymous referees aided us in improving this paper. In particular, in clarifying the relationship of our abstract machine definition to the practical application of code replacement techniques in present-day run-time environments.

Chris Walton is supported by a postgraduate studentship from the EPSRC. Dilsun Kırlı is supported by a postgraduate studentship from the University of Edinburgh. Stephen Gilmore is partly funded by the 'Distributed Commit Protocols' grant from the EPSRC and by the Esprit Working group FIREworks.

Appendix A. Abstract machine definition

A.1. Programs

$$\frac{\begin{array}{l} P = (\bar{D}^k,\ X) \\ (H,\ E,\ ES,\ RS,\ D^1) \Rightarrow (H_1,\ E_1,\ ES,\ RS)\ldots \\ \ldots(H_{k-1},\ E_{k-1},\ ES,\ RS,\ D^k) \Rightarrow (H_k,\ E_k,\ ES,\ RS) \\ (H_k,\ E_k,\ ES,\ RS,\ X) \Rightarrow (H_{k+1},\ E_{k+1},\ ES_1,\ RS_1) \end{array}}{(H,\ E,\ ES,\ RS,\ P) \Rightarrow (H_{k+1},\ E_{k+1},\ ES_1,\ RS_1)} \tag{A.1}$$

A.2. Datatypes

$$\frac{}{\begin{array}{l} (H,\ E,\ ES,\ RS,\ \ \textbf{datatype}\,tn\,\textbf{of}\ \overline{(con,\ \tau)}^k) \Rightarrow \\ \quad (H[p_1 \mapsto \tau^1 \to tn, \ldots, p_k \mapsto \tau^k \to tn] \\ \quad E[tn][con^1 \mapsto p_1, \ldots, con^k \mapsto p_k],\ ES,\ RS) \end{array}} \tag{A.2}$$

Comment: (Rule A.2) 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*. Function types are allocated on the type heap $H[p \mapsto \tau \to tn]$, and entered into the environment $E[tn][con \mapsto p]$.

$$\frac{E(con) = p_1 \qquad H(p_1) = p_2 \to p_3}{(H,\ E,\ ES,\ RS, \textbf{con}\,con) \Rightarrow (H[l_1 \mapsto con], E, ES, (l_1, p_3)\cdot RS)} \tag{A.3}$$

$$\frac{\begin{array}{l} (H,\ E,\ ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ ES, (l_1,\ p_1)\cdot RS) \\ E(con) = p_2 \qquad H_1(p_2) = p_3 \to p_4 \end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{con}\ (con,\ X)) \Rightarrow (H_1[l_2 \mapsto con(l_1)],\ E,\ ES, (l_2,\ p_4)\cdot RS)} \tag{A.4}$$

Comment: (Rules A.3 and A.4) Constructing a datatype value is analogous to applying the constructor function $\tau \rightarrow tn$. For a unary constructor, an argument X of type τ is required. A new constructor value is allocated on the value heap with associated type *tn* in the type heap.

$$\frac{\begin{array}{l}(H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ E(con) = p_2 \qquad H_1(p_2) = p_3 \rightarrow p_4 \qquad H_1(l_1) = con(l_2)\end{array}}{(H, E, ES, RS, \textbf{decon}\ (con, X)) \Rightarrow (H_1, E, ES, (l_2, p_3) \cdot RS)} \tag{A.5}$$

A.3. Values

$$\frac{}{(H, E, ES, RS, \textbf{scon}\, scon) \Rightarrow (H[l \mapsto scon][p \mapsto \tau_{scon}], E, ES, (l, p) \cdot RS)} \tag{A.6}$$

Comment: (Rule A.6) τ_{scon} is the type of the special constant *scon* (e.g. t_int).

$$\frac{E(v) = (l, p)}{(H, E, ES, RS, \textbf{var}\, v) \Rightarrow (H, E, ES, (l, p) \cdot RS)} \tag{A.7}$$

$$\frac{}{\begin{array}{l}(H, E, ES, RS, \textbf{fn}\ (v, \tau_1 \rightarrow \tau_2) = X) \Rightarrow \\ \quad (H[l \mapsto \langle\langle E, v, X\rangle\rangle][p \mapsto \tau_1 \rightarrow \tau_2], E, ES, (l, p) \cdot RS)\end{array}} \tag{A.8}$$

Comment: (Rule A.8) This rule allocates a new closure on the value heap. The closure consists of a copy of the environment, a variable to be bound to the function parameter, and an expression for the body of the function.

A.4. Structured expressions

$$\frac{\begin{array}{l}(H, E, ES, RS, X_1) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ cmap = \{c^1 \mapsto X_2^1, \ldots, c^k \mapsto X_2^k\} \qquad H_1(l_1) = val \\ X_4 = \text{if } val \in \text{Dom}\, cmap \text{ then } cmap(val) \text{ else } X_3 \\ (H_1, E, ES, RS, X_4) \Rightarrow (H_2, E, ES, RS_2)\end{array}}{(H, E, ES, RS, \textbf{switch}\, X_1\, \textbf{case}\ (cmap, X_3)) \Rightarrow (H_2, E, ES, RS_2)} \tag{A.9}$$

$$\frac{\begin{array}{l}(H, E, ES, RS, X_1) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ (H_1, E[v \mapsto (l_1, p_1)], ES, RS, X_2) \Rightarrow (H_2, E_2, ES, RS_2)\end{array}}{(H, E, ES, RS, \textbf{let}\, v = X_1\, \textbf{in}\, X_2) \Rightarrow (H_2, E, ES, RS_2)} \tag{A.10}$$

$$\frac{\begin{array}{l}(H, E, ES, RS, X^1) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \ldots \\ \ldots (H_{k-1}, E, ES, (l_{k-1}, p_{k-1}) \cdots (l_1, p_1) \cdot RS, X^k) \Rightarrow (H_k, E, ES, (l_k, p_k) \cdots (l_1, p_1) \cdot RS)\end{array}}{(H, E, ES, RS, \textbf{record}\, \bar{X}^k) \Rightarrow (H[l \mapsto \{l_1, \ldots, l_k\}][p \mapsto \{p_1, \ldots, p_k\}], E, ES, (l, p) \cdot RS)} \tag{A.11}$$

Comment: (Rule A.11) A record is constructed by evaluating its members $\bar{X}^k$ in left-to-right order (Standard ML record labels are removed earlier in the compilation). The resulting (l, p) pairs are kept on the result stack *RS* until the last one is evaluated. A record $\{\bar{l}^k\}$ is then allocated on the value heap (with a corresponding type on the type heap) to hold the results.

$$\frac{\begin{array}{l}(H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ H_1(l_1) = \{\bar{l}^k\} \qquad H_1(p_1) = \{\bar{p}^k\}\end{array}}{(H, E, ES, RS, \textbf{select}\ (i, X)) \Rightarrow (H_1, E, ES, (l^i, p^i) \cdot RS)} \tag{A.12}$$

A.5. Function expressions

$$\frac{\begin{array}{l}(H[l_f^1 \mapsto \Omega, \ldots, l_f^k \mapsto \Omega][p_f^1 \mapsto \tau^1, \ldots, p_f^k \mapsto \tau^k],\\ \quad E[v^1 \mapsto (l_f^1, p_f^1), \ldots, v^k \mapsto (l_f^k, p_f^k)], ES, RS, X_1^1)\\ \quad \Rightarrow (H_1, E_1, ES, (l_1, p_1) \cdot RS)\\ (H_1[l_f^1 \overset{upd}{\mapsto} l_1], E_1, ES, RS, X_1^2) \Rightarrow (H_2, E_1, ES, (l_2, p_2) \cdot RS) \ldots\\ (H_{k-1}[l_f^{k-1} \overset{upd}{\mapsto} l_{k-1}], E_1, ES, RS, X_1^k) \Rightarrow (H_k, E_1, ES, (l_k, p_k) \cdot RS)\\ (H_k[l_f^k \overset{upd}{\mapsto} l_k], E_1, ES, RS, X_2) \Rightarrow (H_{k+1}, E_1, ES, RS_{k+1})\end{array}}{(H, E, ES, RS, \mathbf{fix}\, \overline{(v, \tau) = X_1}^k\, \mathbf{in}\, X_2) \Rightarrow (H_{k+1}, E, ES, RS_{k+1})} \quad \text{(A.13)}$$

Comment: (Rule A.13) This rule achieves a simultaneous binding of a sequence of function closures (obtained from evaluating $\bar{X}^k$) to the variables $\bar{v}^k$. This is performed by initially allocating a dummy closure $H[l \mapsto \Omega]$ on the heap for each variable. The closure expressions are then evaluated in turn, and the dummy closures are updated to real closures $H[l_{\text{old}} \overset{upd}{\mapsto} l_{\text{new}}]$. Thus, when the body expression X_2 is evaluated, all of the dummy closures will have been updated, and any closure which references another will do so correctly when evaluated.

$$\frac{\begin{array}{l}(H, E, ES, RS, X_1) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS)\\ (H_1, E, ES, (l_1, p_1) \cdot RS, X_2) \Rightarrow (H_2, E, ES, (l_2, p_2) \cdot (l_1, p_1) \cdot RS)\\ H_2(l_1) = \langle\langle E_1, v, X_3 \rangle\rangle\\ (H_2, E_1[v \mapsto (l_2, p_2)], ES, RS, X_3) \Rightarrow (H_3, E_2, ES, RS_1)\end{array}}{(H, E, ES, RS, \mathbf{app}(X_1, X_2)) \Rightarrow (H_3, E, ES, RS_1)} \quad \text{(A.14)}$$

Comment: (Rule A.14) The function application rule applies the function expression X_1 (which evaluates to a closure) to the argument expression X_2. Firstly, both expressions are evaluated. The closure is then obtained from the result of X_1, and the result of X_2 is bound to the variable v in the closure environment E_1. The body of the closure X_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.

A.6. Exceptions

$$\frac{(H[p \mapsto \tau \to \text{t\_exn}], E[excon \mapsto p], ES, RS, X) \Rightarrow (H_1, E_1, ES, RS_1)}{(H, E, ES, RS, \mathbf{exception}(excon, \tau)\, \mathbf{in}\, X) \Rightarrow (H_1, E, ES, RS_1)} \quad \text{(A.15)}$$

Comment: (Rule A.15) This construct introduces the declaration of an exception which can subsequently be raised or handled. The effect of an exception declaration is analogous to that of adding a constructor to a pre-defined datatype named t_exn (compare with Rule A.2).

$$\frac{E(excon) = p_1 \qquad H(p_1) = p_2 \to p_3}{(H, E, ES, RS, \mathbf{excon}\, excon) \Rightarrow (H[l_1 \mapsto excon], E, ES, (l_1, p_3) \cdot RS)} \quad \text{(A.16)}$$

$$\frac{\begin{array}{l}(H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS)\\ E(excon) = p_2 \qquad H_1(p_2) = p_3 \to p_4\end{array}}{(H, E, ES, RS, \mathbf{excon}\ (excon, X)) \Rightarrow (H_1[l_2 \mapsto excon(l_1)], E, ES, (l_2, p_4) \cdot RS)} \quad \text{(A.17)}$$

$$\frac{\begin{array}{l}(H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS)\\ E(excon) = p_2 \qquad H_1(p_2) = p_3 \to p_4 \qquad H_1(l_1) = excon(l_2)\end{array}}{(H, E, ES, RS, \mathbf{dexcon}\ (excon, X)) \Rightarrow (H_1, E, ES, (l_2, p_3) \cdot RS)} \quad \text{(A.18)}$$

$$\frac{(H,\ E,(),\ RS,\ X) \Rightarrow (H_1,\ E,(),\ RS_1)}{(H,\ E,(),\ RS,\ \mathbf{raise}\, X) \Rightarrow \mathbf{halt}\ \ (H_1,\ E,(),\ RS_1)} \tag{A.19}$$

Comment: (Rule A.19) If there are no closures on the exception stack then a raised exception will not be handled. The effect of an unhandled exception is to halt the evaluation of the computation on the abstract machine unless the exception has been raised during the execution of a code replacement operation. That mode of evaluation is described in Appendix B on our modified garbage collection operation. In particular, Rule R1b is relevant here.

$$\frac{\begin{array}{l}(H, E, (l_1,\ p_1)\cdot ES,\ RS,\ X) \Rightarrow (H_1,\ E, (l_1,\ p_1)\cdot ES,\ RS_1)\\ H_1(l_1) = \langle\langle E_1,\ v,\ X_1\rangle\rangle\\ (H_1,\ E_1[v \mapsto (l_1,\ p_1)],\ ES,\ RS,\ X_1) \Rightarrow (H_2,\ E_2,\ ES,\ RS_2)\end{array}}{(H,\ E, (l_1,\ p_1)\cdot ES,\ RS,\ \mathbf{raise}\, X) \Rightarrow (H_2,\ E,\ ES,\ RS_2)} \tag{A.20}$$

Comment: (Rule A.20) 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 A.14).

$$\frac{\begin{array}{l}X_2 = (\mathbf{fn}(v,\ \tau_1 \to \tau_2) = X_3)\\ (H,\ E,\ ES,\ RS,\ X_2) \Rightarrow (H_1,\ E,\ ES, (l_1,\ p_1)\cdot RS)\\ (H_1,\ E, (l_1,\ p_1)\cdot ES,\ RS,\ X_1) \Rightarrow (H_2,\ E,\ ES_2,\ RS_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \mathbf{handle}\, X_1\, \mathbf{with}\, X_2) \Rightarrow (H_2,\ E,\ ES,\ RS_2)} \tag{A.21}$$

Comment: (Rule A.21) This rule ensures that an exception raised in X_1 is handled by X_2 (which is syntactically a closure, as ensured by the equation $X_2 = (\mathbf{fn}(v,\ \tau_1 \to \tau_2) = X_3)$). This amounts to simply applying Rule 8 to X_2 and placing it on the exception stack while X_1 is evaluated. The **raise** rule performs the actual evaluation of the exception handler.

Appendix B. Garbage collection with replacement

$$\frac{}{((),\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} ((),\ \emptyset,\ H_f,\ H_t)} \tag{R0}$$

$$\frac{l \notin \mathrm{Dom}\, H_t \qquad p \notin \mathrm{Dom}\, RM}{((l,\ p)\cdot S, RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (p\cdot FL(H_f,\ l,\ p)\cdot S, RM,\ H_f,\ H_t[l \mapsto H_f(l)])} \tag{R1}$$

$$\frac{\begin{array}{l}l \notin \mathrm{Dom}\, H_t \qquad p \in \mathrm{Dom}\, RM\\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{rep}}) \qquad H(l_{\mathrm{rep}}) = \langle\langle E_1,\ v,\ X\rangle\rangle\\ (H_f \uplus H_t,\ E_1[v \mapsto (l,\ p)],\ ES,\ RS,\ X) \Rightarrow (H_2,\ E_2,\ ES,\ (l_{\mathrm{new}},\ p_{\mathrm{new}})\cdot RS)\end{array}}{((l,\ p)\cdot S, RM, H_f, H_t) \Rightarrow_{\mathrm{gc}} (p\cdot FL(H_f,\ l,\ p)\cdot S,\ RM,\ H_f, H_t[l \mapsto H_2(l_{\mathrm{new}})])} \tag{R1a}$$

$$\frac{\begin{array}{l}l \notin \mathrm{Dom}\, H_t \qquad p \in \mathrm{Dom}\, RM\\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{new}}) \qquad H(l_{\mathrm{rep}}) = \langle\langle E_1,\ v,\ X\rangle\rangle\\ (H_f \uplus H_t,\ E_1[v \mapsto (l, p)],\ ES,\ RS,\ X) \Rightarrow \mathbf{halt}\ \ (H_2,\ E_2,\ ES, (l_{\mathrm{new}},\ p_{\mathrm{new}})\cdot RS)\end{array}}{((l, p)\cdot S, RM, H_f, H_t) \Rightarrow_{\mathrm{gc}} ((),\ \emptyset,\ H_f,\ H_f)} \tag{R1b}$$

$$\frac{l \in \mathrm{Dom}\, H_t}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (S,\ RM,\ H_f,\ H_t)} \tag{R2}$$

$$\frac{p \notin \mathrm{Dom}\, H_t \qquad p \notin \mathrm{Dom}\, RM \qquad H_f(p) = ty}{(p\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (FP(ty)\cdot S,\ RM,\ H_f,\ H_t[p \mapsto ty])} \tag{R3}$$

$$\frac{p \notin \mathrm{Dom}\, H_t \qquad p \in \mathrm{Dom}\, RM \qquad RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{rep}}) \qquad H_f(p_{\mathrm{rep}}) = ty}{(p \cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (FP(ty) \cdot S,\ RM,\ H_f,\ H_t[p \mapsto ty])} \tag{R3a}$$

$$\frac{p \in \mathrm{Dom}\, H_t}{(p \cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathrm{gc}} (S,\ RM,\ H_f,\ H_t)} \tag{R4}$$

References

[1] S. Gilmore, D. Kırlı, C. Walton, Dynamic ML without Dynamic Types, Technical Report ECS-LFCS-97-378, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1997.

[2] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML, rev. ed., MIT Press, Cambridge, MA, 1997.

[3] G. Morrisett, R. Harper, Semantics of memory management for polymorphic languages, Technical report, School of Computer Science, Carnegie Mellon University, 1996; Also published as Fox Memorandum CMU-CS-FOX-96-04.

[4] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, P. Lee, TIL: A type-directed optimising compiler for ML, ACM Conference on Programming Language Design and Implementation, Philadelphia, 1996, pp.181–192.

[5] Z. Shao, An overview of the FLINT/ML compiler, Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97), Amsterdam, The Netherlands, June 1997.

[6] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. Olesen, P. Sestoft, P. Bertelsen, Programming with regions in the ML-Kit, Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, 1997.

[7] N. Benton, A. Kennedy, G. Russell, Compiling Standard ML to Java bytecodes, Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98), Baltimore, 1998, pp. 129–140.

[8] J. Andersson, M. Comstedt, T. Ritzau, Run-time support for dynamic Java architectures, ECOOP'98 Workshop on Object-Oriented Software Architectures, Brussels, July 1998.

[9] W. Taha, T. Sheard, Multi-stage programming with explicit annotations, Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997, pp. 203–217.

[10] M. Shields, T. Sheard, S. Peyton Jones, Dynamic typing as staged type inference, Proceedings of the 25th ACM Symposium on Principles of Programming Languages, San Diego, California, January 1998.

[11] P. Wickline, P. Lee, F. Pfenning, Run-time code generation and Modal-ML, Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 1998, pp. 224–235.

[12] P. Lee, M. Leone, Optimising ML with run-time code generation, Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, May 1996, pp. 137–148.

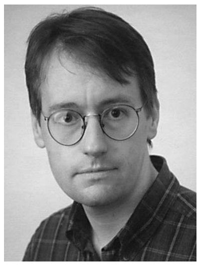
[13] T.M. Chilimbi, J.R. Larus, Using generational garbage collection to implement cache-conscious data placement, Proceedings of the 1998 ACM SIGPLAN International Symposium on Memory Management (ISMM), Vancouver, October 1998.

[14] D. Griswold, The Java Hotspot Virtual Machine Architecture, SUN Microsystems White Paper, March 1998.

[15] T. Kistler, M. Franz, Perpetual adaptation of software to hardware: An extensible architecture for providing code optimization as a central system service, Technical Report 99-12, Department of Information and Computer Science, University of California, March 1999.

Chris Walton is currently working towards a Ph.D. in the LFCS at the University of Edinburgh where he received his B.Sc. His research is focused on the combined areas of code-mobility, code-dynamics, and distributed computing in both Standard ML and Java.

Dilsun Kırlı received her B.Sc. in Computer Engineering from METU, Turkey, and her M.Sc. from the University of Edinburgh. She is currently pursuing research towards a Ph.D. within the LFCS. Her interests encompass foundations of programming languages, mobile computation and program analysis.



Stephen Gilmore received both his B.Sc. and Ph.D. from the Queen’s University of Belfast. He holds a lectureship in computer science at the University of Edinburgh. His interests include functional programming and performance evaluation.