Dynamic ML without Dynamic Types

Stephen Gilmore, Dilsun Kırlı and Christopher Walton
Laboratory for Foundations of Computer Science
Department of Computer Science
The University of Edinburgh

June 15, 1998

Abstract

We describe a variant of the Standard ML programming language which incorporates a facility for the replacement of modular components during program execution. This useful extension of the language builds upon existing compiler technology which permits the separate compilation of modular units of a Standard ML program. Defined naively, this extension would incur performance overheads due to the need to retain typing information at run-time. Here we explain how this cost can be significantly lessened and in some cases eliminated entirely. The essential technical device which we employ for implementation of our extension to the language is a modification of two-space copying garbage collection.

Keywords: modular programming; static type checking; dynamic languages

1 Introduction

Standard ML is a reliable and useful programming language which provides the expressivity of a functional programming language in a setting where the full power of imperative programming constructs is also available. The programmer can choose to work in one setting or the other, supported by a clear distinction between mutable and immutable values. Standard ML has a polymorphic type system and sophisticated modules (called *structures*) which provide flexible encapsulation (due to interfaces called *signatures*) in addition to flexible assembly (due to parametric modules called *functors*).

Standard ML is a strongly typed language. In order to enforce the application of the type-checking phase the language makes 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 activities 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, this useful enforcing of the distinction between the elaboration and evaluation phases also prohibits the modification of programs during their execution. At first sight, preventing the modification of an executing program might seem like an eminently sensible thing to do but increasingly there are more situations where that is not the case. For long-lived computations a programmer might wish to make a minor modification in order to correct a programming error or make an adjustment in order to improve the efficiency of a heavily-used function. Under Standard ML's elaboration-evaluation model neither operation is allowed. Here we show how to regulate the replacement process in order to facilitate updating of program components without interrupting program execution or compromising type safety. We give the Standard ML-like language which allows component replacement the name 'Dynamic ML', reflecting its ability to dynamically replace compiled program code at run-time.

2 A replacement model

In order to facilitate the replacement process, and to indicate a suitable size of component for replacements, we fix on *first-order module-level* replacement. That is, we allow the replacement of signatures by other signatures and structures by other structures, under conditions which we will present later. Functors are compile-time objects which are used to generate structures and so we do not present a replacement model for these. Semantically, module-level replacement is an apt choice because it facilitates the improvement of programs by revision and replacement of data structures in tandem with the creation and access functions which are associated with them. In implementation terms, module-level replacement is supported by the facility to compile Standard ML modules in isolation. Thus an executing program may be upgraded without the need to re-compile it in its entirety. In order that module-level replacement is all that is needed to provide a reasonably comprehensive dynamic code replacement facility, we alter the allowable forms of top-level declaration in Dynamic ML to remove all core-language declarations at top level. We then include a distinguished main program function:

a function `main` in a structure `Main`. In addition we reserve a distinguished structure name, `Install`.

Standard ML deploys inference of static program information in checking the well-formedness of a program during compilation. The inference of types for functions and values in the core language is complemented by the inference of signatures for structures (but not functors) in the modules language. Analogously to the notion of most general type, called the *principal type*, in the core language there is a notion of most general signature in the modules language. The *principal signature* for a structure is the most permissive one, allowing all of the definitions in the structure body to be seen outside. In order to avoid the need to consider the case of structures which do not have signatures we will assume here that all structures have an explicit signature ascription, even if only the principal one.

2.1 Structure replacement

As our running example in this paper we consider the replacement of one implementation of a name table with another which is functionally equivalent but offers improved performance. For simplicity we choose the type of names to be simply character strings. The table type is abstract and in moving from the inefficient implementation to the more efficient one we are in fact replacing an unsorted list with a binary search tree. Both implementations match the `TABLE` signature shown in Figure 1. That is, both implementations provide an abstract type for tables, a declaration of a type of names to be implemented by strings, a constant value denoting the empty table and functions to insert names and then test for membership in the table. Standard ML structures provide dot notation for accessing the components of structures and thus structures named `Table` matching the `TABLE` signature would define types `Table.name` and `Table.table`, a constant `Table.empty`

```
signature TABLE =
sig
  type table
  type name = string
  val empty: table
  val insert: name * table -> table
  val member: name * table -> bool
end;
```

Figure 1: An interface signature for name tables

and functions `Table.insert` and `Table.member`. The implementations are shown in Figure 2 and Figure 3. Matching against the signature (using the `:>` syntax) is *opaque* and thus outside the structure body we cannot make use of the typing information which is known inside the structure body. For example, we cannot make use of the fact that in the first version of the structure tables are implemented as Standard ML lists. This prevents us from applying functions defined on lists (such as `List.rev`) to values of type `Table.table`. Inside the structure body we are free to use list constants such as the empty list, denoted by two square brackets, and list constructors such as cons, denoted by two colons.

After a careful comparison of the two implementations, we should be able to agree that the second implementation could be used as a replacement for the first. A specification-based analysis of the two implementations would judge them to be behaviourally equivalent. Less prosaically, we could think that we would not compute any different results if we had initially built our system using the second implementation of the structure instead of the first. Our notion of structure replacement certainly includes logically undetectable replacements such as this one but it is more lax, additionally allowing replacements which change the observable behaviour of the program under modification. Examples of the latter would include replacing a structure with a version which logged function calls, perhaps in order to extract statistical information about the program's run-time performance or in order to aid with the detection of logical errors in the implementation.

The formal requirement which structure replacements must satisfy can be captured by static type-checking: a replacement `S2` for a structure `S` must match every signature constraint which `S` matched in the original program. In practice, this means that the replacement structure must not omit any functions, types or values which were exported by the structure which it replaces. Functions, values and types which were defined and only used internally may be omitted in the implementation of the replacement structure. For simplicity here, we do not provide a facility for combining replacement with renaming. Structures and signatures replace structures and signatures which have the same name.

2.2 Signature replacement

Signature replacement is a facilitating operation which allows more permissive signatures to replace more restrictive ones. An effect of this can be to make visible functions, types and values which had been hidden by the application of a signature constraint. Replacement such as this is subject to type preservation conditions which constrain the relationship between the

```
structure Table :> TABLE =
struct
  type name = string
  type table = name list
  val empty = []
  fun insert (s, t) = s :: t
  fun member (s, []) = false
    | member (s, h::t) = s=h orelse member (s, t)
end;
```

Figure 2: An inefficient implementation of a name table

```
structure Table :> 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
end;
```

Figure 3: An improved implementation of a name table

signatures. Signature replacement must not cause changes of typing information about visible values and functions. It might be more appropriate to term this operation *signature extension* since it will be most often used to allow the declarations in a structure to be supplemented by others which increase the functionality of any matching structure.

Consider the situation where we replace our `TABLE` signature with one which includes the following function specification:

```
val checkpoint: table * TextIO.outstream -> unit
```

That is, upon being applied to a pair of a table and an output stream such as `TextIO.stdOut` the function will serialize the contents of the table to the output stream and return a unit value (of type `unit`) to signal completion.

We could not replace our previous version of the `TABLE` signature with one extended by the checkpointing function without first upgrading all structures which match this signature to contain an implementation of the function. This would not be visible under the old signature but would become visible when we upgrade the signature to include it. We could then replace client structures of the `TABLE` structure to allow them to make calls to the newly added checkpointing function.

3 An implementation model

Having outlined the types of code replacement which we would like to perform, we now go on to describe a simple implementation strategy for them. One could first consider attaching the compiled image of the replacement code to the program image, reassigning function pointers and invoking garbage collection to remove the old, now unused, code. A moment's thought will be enough to convince us that this cannot work. We would realise that values calculated by the old version of the code would still be live in memory and upon the first application of a replacement function to one of these we would be able to use a data value in a way not allowed by its type. In this way, by fracturing the elaboration-evaluation model, we would allow the programmer to circumvent the helpful static type-checking of Standard ML. This was never our intention.

Instead we must do slightly more work to bring about structure replacement in a type-safe way. We present a new structure together with a replacement mapping which shows how to upgrade from the old representations of data values to the new ones.

3.1 Expressing the replacement operation

Our method of code replacement is intended to be suitable for updating programs where the application programmer has followed good software engineering practice by encapsulating information such as the concrete representation of data structures. This disciplined approach to programming facilitates our replacement of a list by a tree, ensuring that the change is invisible to the users of the `Table` structure. However, this disciplined approach to programming proves to be a disadvantage when we come to consider the problem of describing the replacement of values of the old datatype with values of the new datatype. Specifically, the constructors of the old concrete representation are not visible, due to the encapsulation which is provided by the application of the `TABLE` signature constraint to the `Table` structure in our example.

In order to circumvent this difficulty we could abstract over a `Table` structure which is specialised to implement a name table as a list of character strings. Structure abstractions such as these are Standard ML functors. Given a structure matching the specialised `TABLE` signature the functor body could describe a structure which implements name tables as binary search trees. In addition the structure could contain functions to convert from the types of the given structure to the types of the new. We place the conversion functions inside an `Install` structure and we follow a convention of mapping values from their old representation to their new one using functions which have the same identifier as the type which they update. Such duplicate use of identifiers is possible in Standard ML because the language maintains different name spaces for different categories of identifiers (value constructors, type constructors and record labels in the core language and signature, structure and functor identifiers in the modules language). Figure 4 shows this method of structure replacement encoded as a Standard ML functor. The functions to update values of type `name` and `table` to use the new types are respectively the identity function (mapping `x` to `x`) and an application of the Standard ML Basis library function implementing folding a function across a list with right associativity. When the `List.foldr` function is applied to the insert function for trees and the empty tree it has been specialised to provide a function which maps lists to binary search trees.

The expressive power which Standard ML functors provide is sufficient to allow us to state our wish to replace lists by binary search trees but it would not be sufficient to allow us to subsequently replace these trees with, say, balanced trees. The reason is this: in Standard ML the type expression which appears in a qualification of a signature expression may only refer to type constructors which are in the scope of the signature expression. The nullary

```
functor InstallTable (structure Table: 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: Table.name -> name = fn x => x
    val table: Table.table -> table = List.foldr insert empty
  end
end;
```

Figure 4: A functor which defines the method of replacement for tables

type constructor `string` and the unary type constructor `list` are in scope by virtue of being pre-defined in the language. However, the binary search tree datatype which we defined in the body of the `InstallTable` functor is not pre-defined. Further, it is not exported from the structure which is formed by applying the functor to the old version of the `Table` structure due to the opaque signature matching against the `TABLE` signature. For this reason, we require for Dynamic ML an extended version of Standard ML's `where type` qualification. The extension of this language feature must permit qualifications of the following form in functor headings.

```
functor InstallTable (structure Table: TABLE where
                           datatype table = empty
                            | node of table * name * table) :> TABLE
```

Datatype declarations in Standard ML are *generative* with declarations which are not replications of existing datatypes producing a new type, distinct from all others introduced up to this point. Datatype specifications in signatures are not generative and we need the `where datatype` qualification to similarly not generate a new type name. Its purpose here is to allow us to describe an existing type defined in a structure named `Table` matching the `TABLE` signature. In a language with this degree of expressivity we are able to describe replacement of programmer-defined local datatypes with other types. We now describe how the replacement is effected.

3.2 Executing replacement

Our implementation strategy for effecting code-replacement is based on a modification of two-space copying garbage collection. Before describing the code-replacement operation, it is necessary to understand the basic copying collection algorithm.

In a uniprocessor implementation, the address space of the heap is divided into two contiguous *semi-spaces*. During normal program execution, only one of these semi-spaces is actually used. Memory is allocated in a linear fashion through this semi-space until an allocation fails. 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 live 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 5). Subsequent memory allocations are performed in the new space (*to* space). The role of the two semi-spaces is then reversed for the next garbage collection.

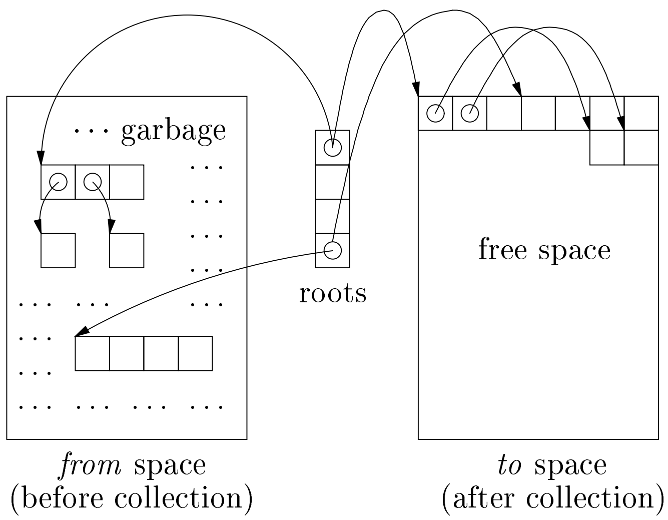

Figure 5: Two-space copying garbage collection

We propose to perform the code replacement operation during garbage collection. A functor, such as the one shown in Figure 4, 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. Replacement is illustrated in Figure 6.

3.3 Failure of installation

3.3.1 No modification due to no match

We must consider two cases where the operation of replacing program components would be judged to have failed. The first case is where there are no components which fit the description of the component which is to be replaced. In our example this would mean that the executing program image contained no compiled representation of a structure name `Table` matching the `TABLE` signature and implementing the type `Table.table` as a list of character strings. A failure of this kind where the compiled program image is not modified at all is classified as a passive failure. The response from the language implementation in this case would be to report a warning to the application programmer stating that the replacement was not installed.

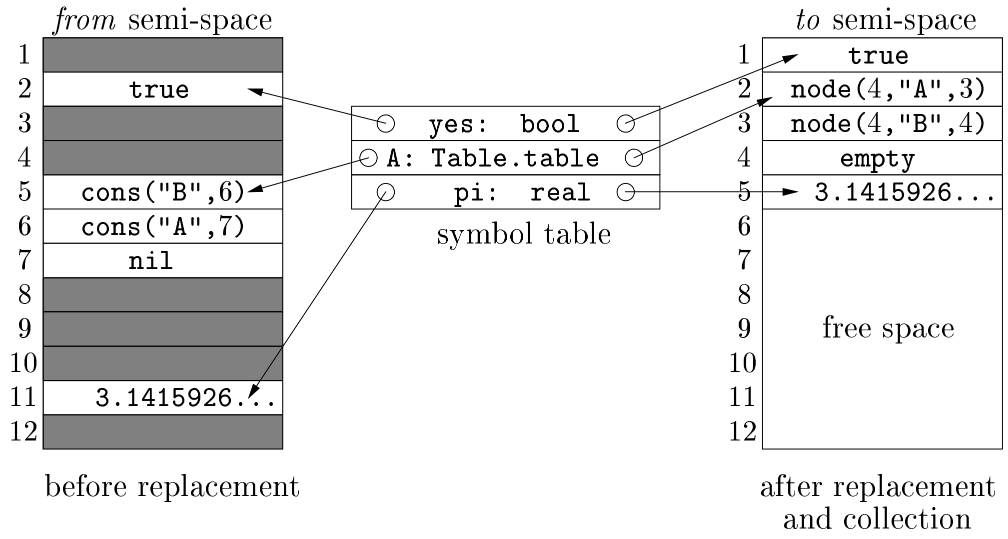

Figure 6: Code replacement with type update

3.3.2 Rollback due to programmer error

The second failure case is a more complex one. We are presenting a modified form of garbage collection operation which executes application programmer code. 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. We can do nothing but weep over non-terminating computations but in the case where an exception is raised we can do more. We would not like the exception-producing function application to halt the execution of our program: we have gone to some trouble here to be allowed to modify our program without interrupting its execution. However, there is no possibility that an exception handler in the program could deal with these errors and recover from them. (We may be introducing new datatypes to which the program had no access.) 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, the old types and we continue with the execution of the old program code. An error report is returned to the application programmer giving as much diagnostic information as possible about the location, nature and cause of the exception.

4 Implementation considerations

4.1 Code distribution

Our eventual intention is to extend the dynamic module-replacement model, described in the previous sections, into a multi-user distributed computing environment. This would considerably enhance the usefulness of the system. For example, it would enable a team of software developers to concurrently introduce updates to a project without requiring a lengthy code-freeze and recompilation stage. Consequently, the choice of garbage collection algorithm was heavily influenced by a desire to avoid unnecessarily complicating this extension.

One promising alternative to garbage collection appears to be region-based memory management [1]. In this scheme the memory consists of multiple stacks each containing values of a particular 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. At first sight, this scheme appears ideal for our purposes as the separation of types into different regions would avoid the need to retain run-time typing information, and would simplify the replacement operation. However, code replacement may have serious implications on the region-inference algorithm, as it will no longer be possible to infer the absolute memory usage at compile time as the program will change over time. Also, the inference algorithm is primarily designed for a uniprocessor system and contains no obvious method of integration into a distributed environment. Furthermore, the source code requires careful profiling to obtain any real performance benefit over traditional garbage collection techniques. Nonetheless, it is possible to combine the allocation of different types into different areas of memory with traditional garbage collection, and this will be investigated thoroughly.

A generational variant of the two-space copying garbage collection algorithm has been successfully integrated into a distributed computing environment in the LEMMA interface [2]. The semi-spaces are divided into two generations: *local* and *global*. The local generation contains data that is only present on the local machine, while the global generation contains data that is shared between machines. Data may migrate from the local generation to the global generation, but not the reverse. Collection of the local generation is exactly as before. Collection of the global generation requires the co-operation of all the machines that have a copy of the shared data. Extending the replacement operation to this generational scheme appears to be relatively straightforward.

We also need to consider how data objects are shared between machines. In the LEMMA interface, a compile-time distinction is made between *immutable* and *mutable* objects. Immutable objects can be freely copied between machines without the need for coherency checking as they will never need to be updated. On the other hand, mutable objects require a coherency checking scheme as an update to an object must propagate atomically to all copies of the object on all machines. Fortunately, only reference types in Standard ML give rise to mutable objects, so relatively little (expensive) coherency checking is required for a typical program constructed in a functional programming style.

Unfortunately, our replacement operation no longer permits us to ignore the coherency checking of immutable objects as they may now be updated during replacement. Consequently, we make the additional distinction between *replaceable* and *irreplaceable* objects for both mutable and immutable types. For example, the signature shown in Figure 1 contains a concrete representation of the type `name` (as `string`) and therefore cannot be replaced. However, the type `table` is abstract and is therefore replaceable. Replaceable objects will require additional coherency checking, though this will not be as expensive as mutable coherency checking as updates are only performed during the replacement (i.e. during garbage collections).

4.2 Performance

Our extension of the Standard ML language would seem to impose some penalties on the run-time performance of our programs. The first penalty which we consider is the overhead which would be incurred by the need to retain typing information at run time. This information is not checked during program execution but it is used when programs are modified by structure replacement. One reason why typing information is shown to be needed in our example is that we must distinguish between lists of character strings which are values of type `Table.table` and those which are of other (perhaps non-abstract) types. As noted in the previous section, tagging of values of replaceable types could be avoided by storing them together in a scheme based on regions. We also observe that only values of datatypes which can be replaced need incur the overhead of retaining typing information at run-time. Values of irreplaceable types can be stored without type information using the well-known techniques for this [3].

A slight run-time penalty is imposed by our dependence on the encapsulation provided by opaque signature matching. Compiler optimisations which could have been performed by exploiting knowledge of the concrete representation of the data structure now may be inadmissible. We believe that it is reasonable to suffer some slight performance penalty for the benefit of ease of software maintenance due to security of representation independence.

A storage penalty is incurred to facilitate signature replacement. The compiler cannot now eliminate so-called 'dead code' (typically functions in structure bodies which are never used or exported). This comes about because of the consideration that a more generous signature might later be applied to the structure, making the invisible functions visible and allowing them to be invoked. In some cases this can be lessened. The `Install` structure which is used in structure replacement can be removed upon successful completion of the code replacement activity.

4.3 Security

A significant concern related to any dynamic language is that the facility to replace code at run-time might leave the system more vulnerable to active attack (perhaps by the introduction of a virus) or passive attack (perhaps by data snooping). The implementation technology should include consideration of this, allowing the prevention of undesired modification perhaps by requiring the user to authenticate any replacement code using well-known public key encryption techniques.

5 A semantic model

Unusually for a practical programming language, Standard ML has a formal definition [4]. The language definition acts as a solid scientific platform upon which may be conducted experiments in programming language design. An alteration to the Standard ML language such as adding first-order module replacement 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" [4, page 42]. Other authors have considered this and argued for the usefulness of a semantic model of memory management in making precise implementation notions such as memory leaks and tail recursion optimisation. A suitable abstract machine model of memory management has been developed [5] and this would form a more suitable setting in which to discuss the Dynamic ML extensions to Standard ML. We intend to provide such a semantics for our extension in a follow-on paper.

6 Related work

Untyped or weakly-typed interpreted languages give the programmer great flexibility in allowing code to be revised while it is executing. For this reason they are sometimes used for the implementation of systems software for distributed computer systems, allowing minor errors to be corrected without disrupting computations which are in progress. However, such languages have inherent problems of inefficiency and insecurity and our interest here has been in bringing the flexibility of code replacement to an efficient and secure language.

This model of first-order module-level replacement for Standard ML programs which has been presented here forms a first step towards the creation of a dynamic variant of Standard ML. Many concurrent or distributed versions of the language already exist [6, 7, 8, 9] but none of these view a replacement model as being crentral to their definition, as we do. Dominic Duggan has recently defined a dynamic variant of Standard ML which has some aims in common with ours [10]. His language differs from ours in that it retains type information at run-time but adds dynamic types without suffering the familiar run-time penalties for these [11]. We hope to explore further the relationship between these two languages.

Acknowledgements

Stephen Gilmore thanks Samantha Osmer for giving the lectures on his course when the early part of this paper was being written. Dilsun Kırlı is supported by a University of Edinburgh scholarship from the Department of Computer Science. Christopher Walton is supported by an EPSRC postgraduate studentship.

References

[1] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[2] 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, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, April 1995.

[3] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

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

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

[6] David C. J. Matthews. A distributed concurrent implementation of Standard ML. In *EurOpen Autumn 1991 Conference*, 1991.

[7] J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 26(6)*, pages 294–305, 1991.

[8] C.D. Krumvieda. *Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming*. PhD thesis, Department of Computer Science, Cornell University, 1993.

[9] Bernard Berthomieu and Thierry Le Sergent. Programming with behaviours in an ML framework: the syntax and semantics of LCS. In *Programming Languages and Systems: ESoP 1994*, number 788 in LNCS, pages 89–104. Springer, April 1994.

[10] Dominic Duggan. A Type-Based Implementation of a Language with Distributed Scope. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in LNCS, pages 277–293. Springer, 1996.

[11] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, October 1993.