Incremental Recompilation for Standard ML of New Jersey

Robert Harper Peter Lee
Frank Pfenning Eugene Rollins

February 1994
CMU-CS-94-116

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-94-02

Abstract

The design and implementation of an incremental recompilation manager (IRM) for Standard ML of New Jersey (SML/NJ) is described. Truly separate compilation is difficult to implement correctly and efficiently for SML because one compilation unit may depend not only on the interface of another, but also on its implementation. In this paper we present an integrated compilation system based on the "visible compiler" primitives provided by SML/NJ that supports "smart recompilation" to minimize system build time in most situations. Large systems are presented as hierarchical *source groups*. An automatic dependency analyzer determines constraints on the order in which sources must be considered. By abstracting away from the specifics of the SML/NJ compiler, the IRM readily generalizes to arbitrary "compilation tools" such as parser generators and embedded languages.

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: Standard ML, smart recompilation, incremental recompilation, separate compilation, modules, dependency analysis

1 Introduction

Large programs are typically built from a collection of independent sources that are repeatedly modified, compiled, and loaded during development and maintenance. The division of programs into separate sources is largely motivated by software engineering considerations such as the need to support simultaneous development by several programers and as an aid to configuration management. Software systems evolve by successive modifications to small collections of sources, after which the system is rebuilt and tested. It is obviously important in practice to cut down on the time required to rebuild the system, particularly since the changes are often of a local nature.

Two main approaches to this problem have been considered. The first, which we shall call *separate compilation*, allows individual sources to be compiled in complete isolation from the other sources, with dependencies between them resolved at "link time". Separate compilation is relatively straightforward for languages with the property that a source may depend only on the interface, and not the implementation, of another source. (This is true of C, for example.) However, Standard ML is unusual in that a source may depend on the implementation of another, rendering truly separate compilation impossible to achieve without penalty (Shao and Appel [12] sketch an approach based on link-time code generation and run-time search to resolve identifier references.) MacQueen argues cogently for this design choice [8, 9]. However, recent work by Harper and Lillibridge [5] and Leroy [7] indicate that it is possible to eliminate implementation dependencies in favor of interface dependencies without significantly compromising the flexibility and expressive power of the SML modules system.

Rather than change the language we consider here an alternative approach, which we shall call *incremental recompilation*, that focuses on providing an efficient integrated compilation mechanism through the use of cacheing and automatic dependency analysis. Our approach is similar to Tichy's "smart recompilation" [14] (and its improvement by Schwanke and Kaiser [11]) in that we rely on a dependency analyzer to determine a set of constraints on the order in which sources must be considered, rather than relying on the programmer to provide an explicit "road map" as in the well-known Unix `make` utility [3]. In order to cope with problems of re-definition of identifiers (what Tichy calls "overloading") and to support shared libraries, we provide a mechanism for structuring collections of sources into what are called *source groups*.

The design of the incremental recompilation manager (IRM) presented here is closely bound to the "visible compiler" primitives of Standard ML of New Jersey, described in a companion paper by Appel and MacQueen [1]. In particular we rely on the SML/NJ primitives for manipulating compile- and run-time environments, and for creating and using compilation units. These primitives provide the fundamental mechanisms needed to build a type-safe IRM that does not rely on hidden internal symbol table mechanisms or the creation of "object files" or "load modules". We gratefully acknowledge Andrew Appel and David MacQueen for their cooperation in the development of the visible compiler primitives in conjunction with the design of the IRM presented here.

In order to properly expose the issues involved we present the design of the IRM in stages, at each step refining the mechanisms and explaining the issues that arise. In Section 2 we introduce the basic evaluation and environment primitives provided by SML/NJ, and consider their use in building large systems viewed as unstructured sets of sources. In Section 3 we refine the evaluation model into compilation and execution phases, and introduce the caching mechanisms required to avoid redundant recompilation of sources. In Section 4 we introduce the notion of a source group as a means of structuring collections of sources and we lift the basic compilation and evaluation primitives from the level of individual sources to the level of systems. In Section 5 we address the problem of dependency analysis for source groups. Finally, Section 6 discusses the generalization

of the IRM to arbitrary "compilation tools" such as parser generators and stub generators.

2 Evaluation of Sources

We begin by considering software systems as unstructured collections of sources, each of which provides a sequence of signature, structure, and functor bindings.[1]

Source code may be presented to the SML/NJ system in several different ways. In order to insulate the implementation from the details of how sources are presented, we introduce an abstract type `source` with the following interface:

```
signature SOURCE = sig
  type source
  val sourceFile   :string -> source
  val sourceString :string -> source
  val sourceAst    :Ast.dec -> source
end
```

The `string` argument to `sourceFile` is the name of a file in an ambient file system. The function `sourceString` presents a source as a string, and `sourceAst` presents a source as an abstract syntax tree (a primitive notion of SML/NJ).

Using the SML/NJ compiler primitives we may define an operation `evaluate` that evaluates a source relative to an environment, yielding an environment:

```
val evaluate :source * environment -> environment
```

The environment argument of `evaluate` governs the identifiers imported by the source, and the result environment represents the bindings exported by the source. (See *The Definition of Standard ML* [10] for a precise explanation of the role of environment in the semantics of SML.)

The following signature summarizes some important operations on environments provided by SML/NJ: [1]

```
signature ENVIRONMENT = sig
  type environment
  val layerEnv   :environment * environment -> environment
  val filterEnv  :environment * symbol list -> environment
  val pervasives :environment
  val topLevel   :environment ref
end
```

The function `layerEnv` combines two environments by layering the first atop the second, shadowing the binding of any identifier in the second that is also bound in the first. The function `filterEnv` eliminates from an environment all but the bindings of the specified list of module-level identifiers (represented in SML/NJ by the type `symbol`). The variable `pervasives` is bound to the environment defining the built-in primitives of the language, and the variable `topLevel` is bound to a mutable reference cell containing the current set of bindings at top level.

To illustrate the use of these primitives we define a `load` primitive (called "`use`" in SML/NJ) that evaluates a single source file relative to the current top-level bindings, and extends the top-level environment with the bindings introduced by the source.

[1] Admitting arbitrary top-level bindings would not affect the development below in any essential way

```
fun load filename =
  let val startEnv = layerEnv (!topLevel, pervasives)
      val deltaEnv = evaluate (sourceFile filename, startEnv)
  in
    topLevel := layerEnv (deltaEnv, !topLevel)
  end
```

Early SML implementations relied on a `load`-like primitive to build large systems — the system is described by a series of `load` operations given in a suitable order reflecting the dependencies between sources. This approach suffers from two major deficiencies. First, it creates pressure on the top-level environment by relying on it both as a temporary "scratch pad" for the interactive loop and as the locus of the software system under development. Second, it relies on the programmer to specify the order in which sources are to be considered. This is problematic because the dependency structure often changes during development. Moreover, overuse of the top-level environment tends to obscure the dependency structure by providing unexpected bindings for free variables.

Our first goal is to eliminate the deficiencies of a `load`-based approach by generalizing the `evaluate` primitive to work with an unordered collection of sources, rather than a single source. The sources of a system are to be presented in a "declarative" style, without regard to the dependencies between them. The generalized `evaluate` primitive must determine a suitable order in which to evaluate the sources, and yield the environment resulting from evaluating each source in turn. Thus we seek to satisfy the following specification:

```
val evaluateSourceSet :source list * environment -> environment
```

For this operation to be well-defined, we must assume that no two sources define the same identifier, and that no source re-defines an identifier already defined by the environment. The former restriction ensures that there is a well-defined order in which the sources may be considered. The latter restriction avoids the ambiguity inherent in a situation where one of two sources makes use of an identifier that is defined both by the other source and the environment of evaluation. We will see how these restrictions can be relaxed through the use of source groups in Section 4 below.

Assuming that these restrictions are met, we may implement `evaluateSourceSet` by evaluating all sources in the set in parallel, synchronizing to ensure that definitions are propagated appropriately. If evaluation cannot proceed, then either there is a circular dependency among the sources in the set, or else some identifier is undefined. Both of these are errors in SML, and evaluation can be aborted. A simple way to implement this parallel evaluation strategy is to make use of first-class continuations [4]. The idea is to associate a continuation with the "undefined identifier" exception that may be invoked to resume evaluation once the identifier becomes defined. Specifically, the evaluator raises the following exception whenever an undefined identifier is encountered:

```
exception UndefinedSymbol of symbol * (environment cont)
```

Evaluation may be resumed by throwing to the packaged continuation an environment that provides a definition for the packaged symbol.

Although conceptually simple, the parallel evaluation strategy appears to be impractical. A pilot study indicated that the memory usage of the above implementation is excessive because most evaluations block early, holding resources that cannot be used until much later. Furthermore, it is difficult to retrofit the SML/NJ compiler to support the interleaving model sketched above. A natural alternative is to determine statically a schedule of the sources based on the dependencies between them, and to evaluate the sources in an order consistent with their mutual dependencies.

The difficulty with this approach is that the scoping mechanisms of SML make it impossible to determine the dependencies between sources without performing what amounts to a form of elaboration. Moreover, this elaboration process must itself be performed in parallel because modules cannot be elaborated without resolving their free identifiers (*e.g.*, to elaborate `open S` we must know the signature of `S`, and we cannot proceed without this information).

Fortunately it is possible to make do with a limited form of evaluation. This is discussed in more detail in Section 5. For the time being we simply postulate the existence of a function `schedule` satisfying the following signature:

```
val schedule :source list * environment -> source list
```

The intent is that `schedule` analyses the given list of sources relative to the given environment, and re-orders the list of sources consistently with the dependencies between them. With this in hand we may define `evaluateSourceSet` as follows:

```
fun evaluateSourceSet (sources, baseEnv) =
  let fun eval (src,env) = layerEnv (evaluate(src,env), env) in
    fold eval (schedule(sources,baseEnv)) baseEnv
  end
```

That is we build the environment resulting from evaluating each of the sources in the order determined by the scheduler, with the input environment for each source being the environment resulting from layering the environments resulting from evaluating the previous sources atop one another.

It is important note that this evaluation in a sequential order introduces spurious dependencies between the sources. The environment in which a source is evaluated is derived from all preceding sources in the schedule, regardless of whether the source actually depends on them. For example, if both `A` and `B` depend on `C`, but neither depends on the other, we may evaluate these three sources either in the order `C`, `A`, `B`, or `C`, `B`, `A`. Suppose the `schedule` function gives us the latter, and that the `B` source is modified. Then the re-evaluation of the system will re-evaluate `A` in an environment reflecting the changes to `B`, even though `A` is insensitive to such changes. We will return to this point in the next section, as it has an affect on the mechanism for incremental recompilation.

3 Incremental Recompilation

The evaluation phase of SML/NJ may be divided into compilation and execution phases.[2] As we have seen in the previous section, evaluation is performed relative to an environment providing both the types and values of the free identifiers of a source. But since the compilation phase requires only the type information, and the execution phase requires only the values, it is natural to split the environment into static and dynamic parts. The compilation and execution phases are linked by a `compiledUnit` which consists of both the generated code for a source and the static environment resulting from elaborating it. These considerations lead to the following specifications:

```
type environment = staticEnv * dynamicEnv
type compiledUnit = staticEnv * codeUnit

val compile : source * staticEnv -> compiledUnit
val execute : compiledUnit * dynamicEnv -> dynamicEnv
```

[2]The compilation phase may itself be refined into parsing, elaboration, and code generation phases, but we shall not make use of this separation.

Using these primitives we may define `evaluate` as follows:

```
fun evaluate (source, (se,de)) =
  let val (se',code) = compile (source,se)
      val de' = execute (code, de)
  in
    (se',de')
  end
```

The environment is decomposed into its static and dynamic parts, the source is compiled relative to the static part, and the resulting code is executed relative to the dynamic parts. The static and dynamic results are then recombined to form the complete result environment.

Having separated evaluation into compilation and execution phases, it is natural to consider cacheing `compiledUnit`s to avoid, where possible, redundant compilation. This may be achieved by simply memoizing the `compile` function. To do so we must define a suitable notion of equality on `staticEnv`s and `source`s.

In principle two sources are equal iff they contain the same SML abstract syntax modulo the names of bound variables. In practice we rely on an easily checked condition that suffices for true equality of sources. When both sources are files, we rely on the ambient file system to define equality of files; otherwise, two sources are deemed inequivalent. This approximation appears to work well in practice, particularly since sources in large systems are usually presented as files.

In principle two static environments are equal iff they govern the same set of identifiers, and ascribe the same type information to each. In practice we rely on a hashing scheme known as an *intrinsic stamp* [1] to compare static environments. Although it is possible for distinct environments to have identical intrinsic stamps, it is highly improbable that such a clash would occur.

The simple approach to memoization sketched above suffers from a limitation arising from the sequentialization of compilation. As discussed at the end of the previous section, the compilation environment for a source may differ solely because previous sources on which it has no dependencies has changed, leading to spurious recompilation. This can be avoided provided that an environment contains sufficient information about how it was constructed so that we may determine which portions of the environment have changed, and using only this information to access the cache.

4 Source Groups and Libraries

Up to this point we have considered programs to be given by an unstructured collection of sources whose namespaces do not conflict. This simplified view is untenable in practice, particularly since we wish to support independent development by teams of programmers and the use of libraries of commonly used programs. In this section we introduce the concept of a *source group* which allows for the hierarchical organization of sources and relaxes the "no redefinition" restriction imposed above.

A group is defined by a set of sources, a list of subgroups, and an optional list of visible symbols.

```
datatype group = Group of
  {sourceSet :source list
   subgroups :group list
   visibles  :symbol list option}
```

A group is evaluated relative to a base environment by recursively evaluating the subgroups relative to the base environment, then layering the resulting environments on top of the base environment, and finally evaluating the list of sources according to some schedule. If there is a list of visible symbols, the result environment is filtered to eliminate all but those bindings mentioned in the list. Notice that we assume that there are no mutual dependencies between subgroups, and that all free identifiers are resolved either within the group or by the base environment.

To avoid unnecessary recompilation during group evaluation, we may use a fine-grained cache currency predicate as we did for evaluation of source sets. In this case, however, we must also check that each symbol imported by a source is defined by the same environment as the cached compilation unit expects. This additional constraint is needed because the same symbol may be defined in different subgroups. If the subgroups are reordered, or filtered differently, then, even if the sources remain unchanged, a symbol may be imported from a different environment than it was when the cached compilation unit was generated.

Source groups provide flexibility in structuring the sources of a system, which is important especially for software development teams. However, there are some issues not addressed by groups, specifically having to do with creating and managing libraries.

1. *Firewalls for currency checking.* Libraries should be considered stable and a library client is unlikely to want to check them repeatedly for currency.

2. *Demand-driven evaluation.* Because of caching of compiled units, compilation of sources is demand-driven for groups. However, *all* sources (or their cached compiled units) in a group are *always executed* in order to conform to the simple environment layering semantics. Since execution changes state, this cannot be optimized significantly.[3] Libraries, on the other hand, are potentially very large, but only a few modules may be needed by any particular client. Evaluating all cached compilation units for the library may thus be impractical.

3. *Unavailability of sources.* In many circumstances the sources for a library will simply be unavailable; in some cases the author of a library might supply only the compiled units.

A library consists of a graph whose nodes are compiled units and whose edges represent dependencies between them. This graph may be used to direct the evaluation of the library to form an environment. A library can be constructed from a group by analyzing the dependencies among sources, compiling them in dependency order, and building a graph from the resulting compilation units. Thus we have the following specification:

```
type library
val makeLibrary :group -> library
```

In most circumstances, a library created from a group becomes persistent and is cached as a library archive file. The graph of compiled units is cached in the file system as a set of pointers to binary files containing the actual compiled units.

To allow for the use of libraries we extend the notion of a group as follows:

[3]Functor bindings would not need to be re-evaluated, but that does not save much, since a functor (essentially) evaluates to itself in one step.

```
datatype group =
   Group of
     {sourceSet :source list
      subgroups :group list
      visibles  :symbol list option}
 | Library of library
```

Libraries are never analyzed for currency; it is assumed that the cached compilation units are always up to date. (Since they have no free identifiers, the compiled code is indepenent of the current environment.)

This approach to libraries is somewhat oversimplified because it assumes that if any portion of a library is to be used, then the entire library must be incorporated. It would be preferable to provide a form of "demand-driven" execution of libraries. This may be achieved by restricting the graph to a given list of symbols prior to executing. This makes sense provided that no library module contains an implicit dependency on any other. Consider the following example in which each structure is defined in a separate source. Structure `A` defines a ref cell called `x` and does not refer to any other modules. Structure `B` initializes the value of `x`. In this case, `A` depends on no other sources, and `B` depends on `A`. The evaluation of a group consisting of `A` and `B` would first evaluate `A` then `B`. If this group is made into a library, and a client only refers to `A`, then `B` will not be evaluated, and the initialization of `x` will not occur. Of course, it is likely that such a group should be reorganized if it is to be made into a library, for example by not allowing structure `A` to be visible.

5 Dependency Analysis

We return now to the problem of statically determining the dependencies between a set of sources, which amounts to determining the set of identifiers imported and exported by each source in the set. As we remarked earlier, this cannot be done without performing at least some of the work performed by the elaboration phase of the compiler. The main complication is that the free identifiers of a source are not syntactically evident. For example, consider the following code fragment:

```
open N
val y = M.x
```

Whether this code imports structure `M` depends on whether `N` defines a structure `M`. Consequently, a dependency analyzer must determine the set of identifiers defined by `N` before it can determine whether the above code fragment imports `M`. This analysis duplicates some of the work performed in compiling each source. In particular, every source must be parsed twice, once by the analyzer and once by the compiler. This overhead does no appear to be prohibitive in practice, and in any case can be avoided only by employing a "parallel" elaboration of the sources in a set, using essentially the strategy sketched in Section 2.

The dependency analyzer of the IRM maps a source group and a base environment to a source-dependency graph:

```
val analyze :group * environment -> sourceGraph
```

A `sourceGraph` can be traversed by to determine a suitable order of evaluation:

```
val evaluateGraph :sourceGraph * environment -> environment
```

We may then define `evaluateGroup` as follows:

```
fun evaluateGroup (g,e) = evaluateGraph (analyze (g,e), e)
```

Sources graphs are constructed using the function `analyzeSource` which constructs the set of symbols imported and exported by a source. This function satisfies the following specification:

```
datatype binding = {symbol :symbol, def :binding list}
val analyzeSource :source -> (symbol -> binding) ->
                   {imports :symbol list, exports :binding list}
```

The type `binding` represents the definition of a module identifier, recording both the symbol being defined and the bindings contained within it. The function `analyzeSource` is memoized to avoid redundant analysis of sources in a set.

Since the sources in a group are not assumed to be in any useful order (except for the subgroups, which must be analyzed before the group itself), we employ a multithreaded strategy similar to the one sketched earlier for parallel evaluation of sources. The function `analyze` calls `analyzeSource` in a separate thread for each source in the group. When the analysis of one source completes, `analyze` enters the exported bindings into a binding table. This table is read by a blocking lookup function which is passed as the second argument to each source analyzer thread. The analysis of each source continues as long as symbols looked up via the lookup function are found. When a lookup fails, the associated `analyzeSource` thread is suspended. A deadlock occurs if some symbol is not defined in one of the sources or if a circular dependency exists among some sources in the group. If a deadlock occurs, an error is reported. After the imports and exports of the sources are determined, a `sourceGraph` can be constructed by creating dependency edges from a source that imports a symbol to the source that exports it.

The dependency analysis of a group is complicated by the fact that the sources in a group may shadow symbols defined in subgroups. Consider the following group:

```
Group {sourceSet = [A, B],
       subgroups = [G],
       visibles  = NONE}
```

Let us assume that both `G` and `A` export a symbol `S`, whereas `B` imports `S`. Care must be taken to ensure that `B` imports `S` from `A` rather than the subgroup `G`. In the IRM this is accomplished by performing a pre-pass on the source set to determine only their exported symbols. The result of this pre-pass is then used to filter the symbols exported by the subgroups for the dependency analysis.

6 Compilation Tools

Thus far we have considered only evaluation of SML sources. However, in practice an SML software system may consist of sources that require other forms of processing. This situation arises, for example, when other languages are implemented in SML, or when tools such as parser and lexical analyzer generators are used. In order to support incremental recompilation for such languages, we have provided a way to incorporate source-processing tools into the IRM.

Our basic assumption is that the ultimate goal of processing a source is the execution of a compiled unit. Thus, a tool can be viewed simply as providing a "compiler" that translates sources into compiled units, which are finally executed in the usual manner. Keeping with the approach

described thus far, we continue to make use of dependency analysis before compilation, and thus a tool must also provide two functions to support the dependency analyzer.

We provide a structure with the following signature for defining and using tools.

```
signature TOOL = sig
  eqtype tool
  val tool :{exportsFn :source -> symbol list,
             analyzeFn :source -> (symbol -> binding)
                          -> {imports :symbol list,
                              exports :binding list},
             compileFn :source * staticEnv -> compiledUnit}
            -> tool
  val exports :source -> symbol list
  val analyze :source -> (symbol -> binding)
                -> {imports :symbol list, exports :binding list}
  val compile :source * staticEnv -> compiledUnit
end
```

New tools are introduced to the IRM by calling `tool`. The functions `exports`, `analyze`, `compile`, and `stamp` each determine the tool for the given source using the `toolOf` function introduced below, and call the function appropriate for that tool. The `exports` and `analyze` functions provide the necessary support for dependency analysis, in the manner described in Section 5. The `compile` and `stamp` functions provide compilation and the support for cacheing of compiled units. The `compile` function is expected to cache its own intermediate results if any, but the IRM caches compiled units.

The signature for abstract type of sources is modified as follows to account for arbitrary compilation tools:

```
signature SOURCE = sig
  type source
  eqtype stamp
  val sourceFile :string * tool -> source
  val sourceString :string * tool -> source
  val sourceAst :Ast.dec * tool -> source
  val toolOf :source -> tool
  val stamp :source -> stamp
end
```

Each source object carries along a tool for processing it. The method for determining whether a cached compiled unit is current with respect to the source relies on an approximation to source equality. The stamp function provides a persistent identifier for a source. Stamp equality approximates source equality. A stamp for a file usually will be constructed from the file's name and modification time, although other schemes are possible.

As an example, consider the following skeletal definition of a tool for processing sml-yacc [13] grammar files.

```
val YaccFileTool =
  let fun generate (src:source) :source = ...
    fun exportsFun src = exports (generate src)
    fun analyzeFun src = analyze (generate src)
    fun compileFun (src, se) = compile (generate src, se)
  in
    tool {exportsFn = exportsFun,
          analyzeFn = analyzeFun,
          compileFn = compileFun}
  end
```

Here, we assume that `generate` implements the parser generator; that is, it reads a grammar from the given source and produces an SML source that implements a parser. In the process, `generate` caches this intermediate output by saving it into a file. Note how `compileFun` calls `compile` to compile the intermediate source into a compiled unit.

The actual IRM implementation allows more dynamic configuration of tools and sources. For example, the list of possible sources is not fixed as indicated by the signature above, but can be extended by registering new types of sources and new tools.

7 Conclusion

We have described an incremental recompilation system for Standard ML of New Jersey. It is faithful to the semantics of Standard ML so that SML programs need not be rewritten in any way in order to be processed by the IRM. The system is computationally parsimonious in that it minimizes recomputation of redundant results. The accuracy of this process is limited only by the conservativeness of the approximations to equality of sources and static environments. Larger systems can be managed, with the ability to structure a software system into a hierarchy of subsystems or libraries, each consisting of a set of sources or compiled units. Finally, the IRM is extensible, allowing for compilation tools other than the SML/NJ compiler to be accommodated.

The design of the IRM allows it to be used in many different ways. In practice, different systems have been defined on top of the basic IRM, each of which implements a particular way of managing incremental compilation. For example, we have defined a structure called `Batch` that defines operations specific to bootstrapping a new version of the SML/NJ compiler, and is convenient for building cross-compilers. A more commonly used example is the structure `Make` that provides a set of operations suitable for use in a typical file-based development environment. This structure provides, among other things, a function

```
val make :string list -> unit
```

which, when given the names of the files (in an arbitrary order) containing the sources for a system, creates a source group from the sources and evaluates it relative to the pervasive environment. Similar functions for managing a system of source groups are also provided.

Over the past few years, several versions of the IRM have been implemented and used daily at a number of sites. The current implementation is richer in some ways than the IRM described here. But, libraries and visible symbols for groups have not yet been implemented. They will be incorporated into the implementation soon. The IRM is provided with the regular distribution of the Standard ML of New Jersey system from AT&T.

References

[1] Andrew W. Appel and David B. MacQueen. Primitives for incremental recompilation of Standard ML. Technical report, Department of Computer Science, Princeton University, Princeton, NJ, November 1993.

[2] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[3] Stuart Feldman. Make–a program for maintaining computer programs. *Software Practice and Experience*, 9(3):255–265, March 1979.

[4] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, ?(?):?–?, ? 1993. (To appear. See also [2].).

[5] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-first ACM Symposium on Principles of Programming Languages*, pages ?–?, Portland, OR, January 1994. (To appear.).

[6] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS–LFCS–86–2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[7] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland.* ACM, January 1994.

[8] David MacQueen. Modules for Standard ML. In *1984 ACM Conference on LISP and Functional Programming*, pages 198–207, 1984. Revised version appears in [6].

[9] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[10] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[11] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988.

[12] Zhong Shao and Andrew Appel. Smartest recompilation. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 439–450, Charleston, SC, January 1993.

[13] David Tarditi and Andrew W. Appel. ML-YACC, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

[14] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.