Separate Compilation for Standard ML*

Andrew W. Appel†
Princeton University
appel@princeton.edu

David B. MacQueen
AT&T Bell Laboratories
macqueen@research.att.com

Princeton University CS-TR-452-94
March 15, 1994

Abstract

Languages that support abstraction and modular structure, such as Standard ML, Modula, Ada, and (more or less) C++, may have deeply nested dependency hierarchies among source files. In ML the problem is particularly severe because ML's powerful parameterized module (functor) facility entails dependencies among implementation modules, not just among interfaces.

To efficiently compile individual modules in such languages, it is useful (in ML, necessary) to infer, digest, and cache the static environment resulting from the compilation of each module.

Our system provides a simple model of compilation and linkage that supports incremental recompilation (a restricted form of separate compilation) with type-safe linkage. This model is made available to user programs in the form of a set of internal compiler modules, a feature that we call the "visible compiler." The chief client of this interface is the IRM incremental recompilation manager from CMU.

1 Interfaces and Implementations

Modular languages, such as Ada and Modula, distinguish between *interface modules* and *implementation modules*. In a weaker sense even C and C++ support such a distinction by organizing source code in .h and .c files. Compilers for these languages support *separate compilation* in which the compilation of each implementation module requires the examination of those interface modules that it imports or exports, but rarely, if ever, the examination of another implementation module.

When the *dependency graph* of interfaces is shallow, it's reasonable to re-parse all the interfaces imported by each compilation unit when any imported interface is modified; this is what C compilers do with Unix `make`[7].

There are still some problems, however:

- The dependency DAG of just the interface modules can be quite deep: some files in the OI library[5], a well-known C++ library for building X-based user interfaces, bring in about 20,000 lines of header files [13], which can be time-consuming to process.

- Sometimes many files depend on a particular module, so we want to avoid unnecessary recompilations when that module is changed.

- Languages may, for various reasons, allow dependencies *between implementation modules*. Having one implementation module dependent on other *implementations* leads to a deeper dependency DAG, causing timestamp-based *make* algorithms to do large amounts of unnecessary recompilation.

Standard ML is an example of a modular language that allows inter-implementation dependencies — in this case, to support a powerful *functor* (parameterized module) facility (see section 2). Ada [4] and Modula-3 [20, 9] have much more limited forms of parameterized modules, called *generics*. As we will explain in section 2, Ada generics also have nontrivial inter-implementation dependencies. Modula-3 generics have restrictions that avoid this problem, but yield a less powerful and less convenient module system. The *functor* facility of Standard ML is so powerful and convenient that it is heavily used; so a heavy-duty approach to its separate compilation is justified.

Another source of inter-implementation dependencies is in-line expansion of functions across module boundaries—whether performed in response to an *inline* pragma (as in Ada), or automatically by an optimizing compiler for *any* language. The system we describe here does not deal with dependencies of this sort, though it could in principle.

*To appear in the SIGPLAN '94 Conference on Programming Language Design and Implementation.

†Supported in part by NSF grant CCR-9200790.

Some of the separate compilation issues enumerated above have been addressed before: Adams *et al.* [1] review (and measure) several replacements for timestamp-based recompilation: an interface can be hashed into a stamp, so that trivial modifications to source code don't necessarily change the hash—and unnecessary recompilations are avoided.

Some Modula compilers[8, 12] create "digested" versions of interface files to speed the importing process. In ML, an export interface is derived by semantic analysis of both interface *and implementation* modules, making it *especially necessary* to use digested interface descriptions, and not just re-elaborate the source code of imported files. The implementation modules are just too big to re-elaborate for each client that imports them.

In this paper we give a "compiler's-eye view" of the separate compilation mechanism as a set of primitives to be used by a "make" system, and we describe the implementation of those primitives. In section 8 we give an overview of a major *client* of these primitives, an incremental recompilation manager, source-dependence analyzer, and library manager called *IRM*. A companion paper[15] gives a full description of this utility.

Our new separate compilation mechanism for Standard ML, implemented in the *Standard ML of New Jersey* compiler,

- allows incremental recompilation in a language with inter-implementation dependencies;
- provides a uniform compilation and linkage environment so that interactive compile-and-execute commands are compatible with separately compiled modules;
- uses *static environments* (analogous to symbol tables) and *dynamic environments* (playing a role similar to linkage tables) that are first-class values and can be composed in an elegant, side-effect-free manner;
- infers the exact export interface of an implementation module, and hashes this into an *intrinsic stamp* for precise recompilations (see section 5);
- enforces type-safe dynamic linkage without requiring "link-time type checking;"
- provides an architecture for type-safe usage of internal components of an interactive system by its own clients (see section 7);
- can be generalized to support cross-module optimization, which (in any language) leads to inter-implementation dependencies.

To infer full interface descriptions in a language with separately compiled higher order modules is not at all a trivial thing, and we believe we are the first to do this. (In existing Ada or Modula-3 systems, the generics cannot be fully compiled separately from their arguments.)

Cutoff Recompilation

What we have achieved, however, is better termed an *incremental recompilation* system. It is not always possible, in our system, to compile a module *in vacuo*, without seeing all of the modules it imports. But if any module is recompiled without changing its export interface, we avoid recompilation of the dependent modules.

Given a set of interfaces $H_1, ..., H_n$ and corresponding implementations $M_1, ..., M_n$, *classical separate compilation* allows the compilation of any module M_i with the examination of some subset $\{H_j\}$ of the interfaces, but without examining the text of the other implementations. Classical separate compilation is not possible for ML, or any language with inter-implementation dependence.[1]

Having once compiled a set of implementations M_j, *recompilation* allows a module M_i (dependent on the implementations M_j) to be compiled without (in the same "session") compiling the M_j again.

If one of the modules M_j is altered—in a way that does not affect its exported interface—and recompiled, *cutoff recompilation* [21] (we use Adams's terminology [1]) allows the previously compiled M_i (dependent on M_j) to be used without recompilation.

Finally, *smartest recompilation*[23] allows M_i to be compiled even without the examination of M_j (or any of the H_j!); the linker must then be made smarter so that it can verify hypotheses about H_j and M_j made during the compilation of M_i. But smartest recompilation has not yet been demonstrated feasible for higher-order functors.

In this paper we will describe a *cutoff recompilation* system, not a *classical* or *smartest* system.

2 Standard ML modules

The module system of ML, like those of Modula and Ada, distinguishes between *interfaces* (called *signatures* in ML) and *implementations* (called *structures* and *functors*). However, in ML the interface of a structure is often only partially described by an explicit signature[2].

A signature in ML specifies what type elements of a structure will be visible to client structures, but un-

[1] In ML the user can always decouple implementations from each other using functors. This is helpful to the user, when true separate compilation is desired; for example, when a client module must be compiled independently of the modules it imports.

[2] In fact, the signature is optional, and can be inferred if it is not specified.

```
signature PARTIAL_ORDER =
  sig
      type elem
      val less : elem * elem -> bool
  end

signature SORT =
  sig
      type t
      val sort : t list -> t list
  end

functor TopSort(P : PARTIAL_ORDER) : SORT =
  struct
      type t = P.elem
      fun sort(l) =  ... P.less(x,y) ...
  end

structure Factors : PARTIAL_ORDER =
  struct
      type elem = int
      fun less(i,j) = (j mod i = 0)
  end

structure FSort : SORT = TopSort(Factors)
```

Figure 1: Transparent signatures

like a Modula interface, it does not render these elements opaque. ML's "transparent" signature matching is designed to support parameterized structures (functors), whose parameter signature is a *partial* interface description that will be instantiated by actual parameter structures containing different types, which must be propagated through the functor. For example, the structure *FSort* of figure 1 will be of use only if it is known that *FSort.t* is the same as *int*. However, this is not specified in the signature *SORT*; nor should it be, for then it would not be possible to apply *TopSort* to partial orders over other types.[3]

Consequently, to compile a client of *FSort* one must first inspect the *implementation module*
`structure FSort : SORT = TopSort(Factors)`
and not merely the (partial) interface, *SORT*.

More generally, to compile one implementation module it may be necessary to inspect the implementations of all the modules it imports (directly or indirectly) to determine the definitions of exported types that are not fully described in signatures.

This contrasts with the situation in Modula 2 and Modula 3, where it is necessary to inspect only (the transitive closure of) the *interface specifications* of the imported modules. The generic interfaces and modules of Modula 3 do not change this situation much, because they can only be instantiated at the top level. So an implementation module A that is an instance of a generic module G is an example of an inter-implementation dependency; but no other implementation module can depend on the implementation of A (only on its interface), so there is never a nontrivial chain of these dependencies. However, this restriction does limit the utility and convenience of Modula-3 generics.

In Ada, an instance of a generic package can occur as a submodule in a package body, and this causes a dependency between that package body and the generic package that it instantiates. Importers of a package that exports inline procedures also depend on that package's implementation. So Ada also is subject to inter-implementation dependencies, though they arise from different mechanisms from those applying in Standard ML.

The solution used in Ada and Modula-3 is to view generic instantiation as a kind of source-language expansion. This is not really "separate compilation" of generic modules themselves, since only their instances are truly compiled. In ML, a functor is type-checked and compiled into machine code independently of its applications.

Though this is a very significant advantage of the ML module system, it does come at the cost of inter-implementation dependences. In section 10 we discuss alternative module systems and programming styles for ML that avoid this dependence. These alternatives can significantly weaken the original Standard ML module system, so our separate compilation system deals directly with inter-implementation dependence.

3 ML compilation model

The original idea of an ML compilation unit was a single "closed" signature, structure, or functor definition (having no free references to external structures). For various reasons this approach proved inadequate (though it can be patched, as we describe in section 10), so for many years there was been no agreed-upon notion of separate compilation.

The first ML compilers were interactive systems, in which successive declarations were evaluated relative to a "top level environment," augmenting that environment with new bindings. Even the ML module system (signatures, structures, and functors) has been used in this way. Hence, separate compilation was not a critical requirement for the design of the module system, which mainly addressed issues of parameterization, program composition, and name spaces.

How, then, are ML programs to be broken up into

[3] One can expand the result constraint SORT of TopSort and include a sharing constraint equating t to P.elem; but this breaks down with higher-order functors. See also section 10.2.

"compilation units?" We have taken the liberal approach that a compilation unit is a single source file containing any sequence of ML declarations: signatures, structures, functors, values, types, etc.[4] There is no special syntax that delineates compilation units: we use the same module language described in *The Definition of Standard ML*[19].

Thus, all existing ML programs are automatically "upward compatible" with the new separate compilation system.

Evaluation interface

ML source code is evaluated in an environment mapping names to types and values, and produces new bindings of names to types and values—a new environment:

$$\begin{aligned} env: &\quad name \rightarrow type \times value \\ evaluate: &\quad source \times env \rightarrow env \end{aligned}$$

To distinguish between compilation and execution, we factor *evaluate* into *compile* and *execute*, and at the same time we factor *env* into *static* and *dynamic* parts. We use *persistent identifiers*, or *pids* as a form of intermediate name.

$$\begin{aligned} env &= statenv \times dynenv \\ statenv: &\quad name \rightarrow type \times pid \\ dynenv: &\quad pid \rightarrow value \\ compile: &\quad source \times statenv \rightarrow statenv \times codeUnit \\ execute: &\quad codeUnit \times dynenv \rightarrow dynenv \end{aligned}$$

Thus *evaluate* is the appropriate composition of *compile* and *execute*. But note that having decoupled these phases, we can perform a chain of *compiles* at one time, and the corresponding chain of *executes* can be done much later.

The values exported from ML modules are dynamic run-time values, such as function closures and run-time constructed data—*not* merely static linkage addresses. It is necessary to *execute* an ML module even to produce its export fields. In a more conventional model of linking (in Ada, Modula, C), all imports and exports exist as static addresses even prior to the execution of any module. The conventional model requires that all variables be initialized to constant values (though they may subsequently be "re-initialized" by the start-up code of a module). This cannot work in ML, where many kinds of values (such as function closures created from the application of higher-order functions) have no reasonable static "default" initialization. Furthermore, "variables" in ML do not really vary, so they cannot be re-initialized.

The *dynenv* maps internal "names" (*pids*) to runtime addresses. Thus, it serves the same purpose as the link-editor table of the conventional model. But it is produced only as the modules execute: the exported fields of a module are produced by, and do not exist prior to, the execution of the module's *machine code*. (Note that this precludes cyclic dependencies among modules in ML.)

The *compile* function returns a pair of a *statenv* and a *codeUnit*, which together we call a *compiledUnit*. The *codeUnit* breaks down into three components: *code*, the machine code for the unit; *imports*, a list of pids designating the imported values; and *exports*, another list of pids designating the exported values.

$$\begin{aligned} compiledUnit &= statenv \times codeUnit \\ codeUnit &= code \times imports \times exports \end{aligned}$$

With further processing of external references in the *statenv* (as described in section 4), the *compiledUnit* can be written (by the Incremental Recompilation Manager) to a binary file.

Linkage and execution: The *code* of a compilation unit implements a (fully closed) function mapping a vector of import values to a vector of export values. It has *no* external references that require link-editing; *all* external references are obtained from the *imports*, which are passed as arguments to the machine code when it is executed. Individual functions exported by the code will generally be *closures* constructed during the execution phase, and these run-time-created closures hold the imported external references. This has always been the case in Standard ML of New Jersey[3], though our current use of *pids* is more sophisticated than our previous use of "lvar-numbers," which were not "persistent" (lvar-numbers did not retain their meaning from one ML process to another, and thus were useless for separate compilation).

The interactive read-eval-print loop maintains a global environment (*env*) comprised of a *statenv* and a *dynenv*. After compiling a compilation unit—yielding a *compiledUnit*—the interactive loop looks up each of the *imports* pids in the *dynenv*, producing a vector of values. Then it applies the *code* to this vector, producing a vector of export values. Then the export values are bound to the corresponding elements of the *exports* pid list to form an new *dynenv* matching the exported *statenv*. This coordinated pair of static and dynamic environments constitutes the *exported environment* of the compilation unit. This incremental exported environment is then layered onto the global environment.

In a separate compilation system, we should not be so eager to execute a unit right after compiling it. Instead, we will save the *compiledUnit* in a "bin" file for

[4] We recommend (and the IRM compilation manager requires) that separately compiled units contain structures, functors, and signatures, but not top-level values and types.

later use, retaining just the *statenv* as input for further compilations.

But since the interactive loop uses the same *compiledUnit* interface as the separate compilation system, separately compiled modules can be loaded into an interactive system.

The following simple example illustrates the compilation and linkage process, using value declarations instead of module declarations:

Source:

```
val a = x+y
val b = x+2*z;
```

Compilation environment: $\langle s_c, d_c \rangle$

$$
\begin{array}{rcl}
\text{static}: & & \\
s_c = & \{ & x \mapsto \langle \text{int}, \text{pid}_x \rangle,\ y \mapsto \langle \text{int}, \text{pid}_y \rangle, \\
 & & z \mapsto \langle \text{int}, \text{pid}_z \rangle \} \\
\text{dynamic}: & & \\
d_c = & \{ & \text{pid}_x \mapsto 3,\ \text{pid}_y \mapsto 4,\ \text{pid}_z \mapsto 5 \}
\end{array}
$$

Compilation results:

$$
\begin{array}{rcl}
\mathit{statenv} & = & \{a \mapsto \langle \text{int}, \text{pid}_a \rangle, b \mapsto \langle \text{int}, \text{pid}_b \rangle\} \\
\mathit{code} & = & \lambda\langle x, y, z \rangle.\langle x + y, x + 2 \times z \rangle \\
\mathit{imports} & = & [\text{pid}_x, \text{pid}_y, \text{pid}_z] \\
\mathit{exports} & = & [\text{pid}_a, \text{pid}_b]
\end{array}
$$

Execution:

$\mathit{dynenv} = \{\text{pid}_a \mapsto v_a,\ \text{pid}_b \mapsto v_b\}$
where
$\langle v_a, v_b \rangle = \mathit{code}(\langle d_c(\text{pid}_x), d_c(\text{pid}_y), d_c(\text{pid}_z) \rangle)$

4 Making binary files

For each compilation unit, our compiler produces a *compiledUnit*: a tuple of code, pids (import, export), and (incremental) static environment. To do separate compilation (or recompilation) these parts must be written to a binary file for later use in another compilation.

It is easy to write machine code to a binary file; each machine has a well defined representation of machine instructions as sequences of bits [25]. For pids we use 128-bit integers which are also easy to write to files.

But our static environments are complex pointer data structures with no "easy" representation as bit strings. Our representation of static environments [17] (for structures, functors, signatures, functor signatures, types, datatypes, values, fixities, symbol tables, etc.) uses 36 different datatypes, many of which have several variants—there are a total of 115 variants of these types, with a total of 193 record fields. It is not a trivial matter to write a linked structure over these types into a file.

Therefore we use a "pickler," a program integrated with the garbage collector of our system that writes a "pickled" version of an arbitrary data structure to a file, and can read it back in again [20, page 80].

Even so, we found that our internal representation of static environments was not suitable for pickling without some modifications:

- Static environments should be self-contained and portable. This means that they should not have pointers into global data structures, nor should they contain function closures because this would complicate cross-compilation for a different architecture.
- Export environments should not be affected by internal, unexported types and structures—we describe the problem and its solution in section 5.
- The static environment of a compilation unit may contain pointers into the static environment of other units on which it depends.

Managing sharing

The exported static environment for a unit points right into the middle of the static environments of other units. The pickler would happily traverse such pointers, but then the binary file will contain a private copy of a shared entity. Later, when the static environment is read from the binary file, it will include a new copy of the (previously shared) entity. In the worst case, the DAG of references between environment components will be converted into a tree, leading to exponential blowup both of binary files and of core memory.

To solve this problem, we "dehydrate" the exported environment before writing it to a file: that is, we identify external pointers and replace them by stubs. Later, after reading the binary file, we will "rehydrate" the environment by turning the stubs back into pointers. This is the static linkage process.

There are two problems to solve in designing the dehydration algorithm:

1. How can the dehydrater tell which things are "external" pointers into structures shared with other bin files, since all pointers in core look the same?
2. Given a "stub," how can the rehydrater find the real in-core pointer with which to replace it?

Every "significant" object in a static environment (module (structure or functor), signature, or type constructor) has its own "stamp," composed of a persistent identifier (pid) and a small integer. All objects created

in one compilation unit will have the same pid but different integers. The dehydrater traverses the static environment exported by current compilation unit; whenever a stamped object is found whose pid belongs to another compilation unit, this object is "external" and can be replaced by a stub containing only the stamp. This is the solution to problem 1.

But later, after the dehydrated environment has been read from the bin file, how can the rehydrater replace the stub with the real pointer?

Each source file is compiled with some "context" environment made from the combination of the export-environments of all the imported modules. Any external object must live somewhere within the context environment. But the environments are not indexed by "stamp;" they are mappings from source-language symbols to objects containing stamps. An exhaustive search of the context environment would find the object containing any given stamp, so in principle the context environment is sufficient to rehydrate.

We make this more efficient by constructing, for each environment, the index mapping from stamps to objects. Each environment now comprises two mappings: from symbols to objects, and from stamps to objects. *Static environment* is an abstract type within the compiler, with operations for construction, layering, lookup, and so on; *Indexed static environment* matches the same specification but also supports lookups by stamp. It is possible to coerce either kind of environment into the other, either by constructing the index or by throwing it away. Within the elaboration of a single compilation unit, non-indexed environments are used (for efficiency) because lookups by stamp are not needed; IRM uses only indexed environments and applies coercions at the appropriate points.

For the rehydrater to work, it must be given a context environment containing all the external stamps that might be found in the current compilation unit. Fortunately, this context is simply the indexed version of the static environment originally used in compilation of this unit. For compilation, IRM constructs context environments by layering the exported *statenvs* of imported units; for rehydration, it just reproduces an equivalent layering. When indexed environments are layered, both the symbol mappings and the stamp mappings are combined separately; this is more efficient than layering the symbol mappings and then re-indexing.

The dehydration and rehydration processes are extremely similar: both traverse a static environment, replacing certain leaf-like nodes (external-pointers become stubs, or vice versa). A single program, with a dehydrate/rehydrate parameter, accomplishes both tasks.

Dehydration/rehydration is done in a nondestructive, functional style: the environment is copied. Copying is quite efficient in SML/NJ (because of fast allocation and fast garbage collection) but one problem is that DAG structures can turn into tree structures (with a big blowup in memory use) unless one is careful. This is not trivial in ML, which has no notion of pointer identity; nor, for that matter, in any language with copying garbage collection, where pointers cannot be used as hash-table indices. But the stamps of internal objects (at least for those objects "significant" enough to have stamps) can serve as indices into temporary tables to ensure that sharing of pointer structure is preserved. Hash-consing would work even better, but we don't have that.

The most troublesome and time-consuming part of the implementation was the detailed work on identifying external sharing (with stubs) and preserving internal sharing (DAG structure). This is in part because there are so many cases to handle: a set of mutually recursive functions had to traverse, and preserve sharing in, all the 115 variants in 36 data types.

Summary: to compile one source file, a context environment is built, and the compiler produces the *compiledUnit* described earlier. The static-environment (*statenv*) part of the *compiledUnit* is saved for use in further compilations *now*, and a dehydrated copy is written to the binary file (along with the *codeUnit*) for use in other compilations *tomorrow*.

On another day, an equivalent context environment is built; the binary file is read back in; and the *statenv* is rehydrated using the context environment.

The rehydrated environment will then be combined with the exported environments of other modules to form contexts for loading other binary files, or compiling other source files.

5 Type-safe linkage

Once the modules have been compiled (or recompiled) they must be linked together. A *classical linker* resolves names that refer to machine code addresses and global data addresses (with link-time static allocation). A classical linker does not maintain any type information, so it is up to some other part of the programming environment (typically the programmer) to ensure that the modules linked together are compatible.

For example, a C programmer may neglect to include some .h/.c file dependency in the makefile. Then when a .h file is altered to change the parameters of some procedure f, a client module calling f with the old arguments might not be recompiled. This error will not be detected by the linker.

A type-safe linker, on the other hand, ensures that the compiled implementations it is linking together have a consistent view of their interfaces[14]. Naturally, we have implemented a type-safe linker. Rollins

has implemented an automatic dependency analyzer whose output is used to control order of evaluation, preventing "makefile bugs" [15]. If the dependency analyzer is not used, or if it should fail, our linker will detect any inconsistency.

Recall from section 3 that the linkage of dynamic values is not by names, but by "pids" which label imported/exported values. An exported pid could be constructed in several ways:

1. The pid could be a simple "timestamp" (augmented with host and process identifiers to ensure uniqueness). This will lead to unnecessary recompilations; when a module is edited and recompiled without changing its interface, a new timestamp (pid) will be generated, and the dependent modules must then be recompiled.

2. The pid could be made by hashing source code (exclusive of comments) of the module together with the pids of the imported interfaces. This approach was taken in the Cedar/Mesa system[14]. Trivial changes to the source code (editing the comments, for example) will not cause the pid to change, and then the dependent modules will not need recompilation.

3. The pid can be a hash of the exported static interface [22, 12]. Then *any modification of the implementation module that does not affect its export interface* will yield the same pid. Bugs can be fixed, and algorithms can be changed, without causing a cascade of recompilation.

Either of the latter two kinds of pid can be called *intrinsic* because they are independent of (time, place) context. The last alternative is obviously preferable. We use intrinsic pids, though our system would accommodate timestamps.

A system using intrinsic pids will naturally use *cutoff* recompilation—avoiding many unnecessary recompilations. Cutoff recompilation can be implemented [21] by comparing the "new" compiler output (describing the export interface) with the previous one; but with the use of intrinsic pids, it is unnecessary to save (or compare) the previously compiled version because we compare hash values and not full environments.

There is always the possibility of collision: two different *statenvs* could hash to the same value. Because we use a good hash function (a CRC of 128 bits) this should happen only once for every 2^{128} pairs of *statenvs.* A very large software system would have perhaps 2^{13} pids, or 2^{26} pairs, so the probability of collision is 2^{-102}. But if this is not good enough, IRM can use intrinsic (hash) pids for software development, and switch to timestamp-based pids for pre-production verification.

Intrinsic pids

The more precisely the *intrinsic pids* express *only* the exported interface, and the less they are dependent on other aspects of the implementation modules, the fewer unnecessary recompilations will be done.

The intrinsic pids are computed by hashing the exported static environment. We do a prefix-order traversal of the *statenv*. For each node, of some datatype with n variants, we "emit" an integer between 1 and n depending on the variant. We take appropriate care to avoid doing a tree traversal of what is actually a DAG. When we come to a pid (either internal or imported from somewhere else and here being re-exported) we emit the pid as an integer.

This produces a sequence of integers, which we then hash by a CRC algorithm to produce the export static pid for this compilation unit. The export pids of the k dynamic objects produced by this unit are derived from the static pid by adding 1 through k respectively to the export static pid.

Note, however, that the static environment contains all the entities to be exported; these entities contain their own pids; these pids must be derived from the result of the hash; and yet these entities are traversed in order to compute the hash.

This paradox is resolved by having the elaboration (semantic analysis) phase of the compiler give all entities "provisional" pids. These are grist for the hasher, and they are replaced by "real" pids after the CRC is computed.

In the course of a compilation, many substructures and types are created with provisional pids, but only some of these are exported. The hasher should not hash the provisional pids directly, for this will make the hash value dependent on the number of hidden internal structures and types used in the implementation of the module. Instead, it must "alpha-convert" them: the hasher keeps a record of which provisional pids have been seen, and uses the hash value of n for the nth distinct pid seen thus far.

Hashing, and replacement of provisional pids, are done before the dehydration phase described in section 4.

Performance Look how many passes we are taking over the export environments!

- Hash the environment to compute intrinsic pids;
- Dehydrate: replace external references with stubs;
- Later, rehydrate: replace stubs with external references.

We measured the time taken by these phases in a 32-minute compilation of 65,000 lines.[5] The mea-

[5] on a DECstation 5000/240

sured time for hashing was 0.0 seconds (modulo clock-tick resolution 0.01 second) and for dehydration/rehydration was 20 seconds (i.e., 1% overhead). We believe this 20 seconds could be very substantially reduced by replacing some of the linear searches we use to identify previously seen nodes (to preserve DAG structure).

Static pids

The *imports* pid list of a unit expresses the dynamic dependences of the unit: what imported values are required for execution. It would be nice if this list completely completely summarized the dependences of the unit on its environment. But some dependences have no dynamic component: type and signatures have only a static effect. Thus, to implement proper *cutoff recompilation*, IRM needs more information.

To characterize a unit's exported static information, it is sufficient to hash the exported *statenv* into a single *static-pid*. In the binary file for each unit M, the IRM system writes the *static-pids* of all imported units and the exported *static-pid*. When the static exports of a compilation unit change, its new *static-pid* will be different. Then, when examining a dependent binary file for currency, IRM will be able to detect that it imports some *static-pid* that is no longer exported anywhere.

It is IRM's responsibility to manage static-pids. IRM, not the compiler, calculates the list of imported units and assembles the corresponding list of static-pids. To make the compiler keep track of imported units would greatly complicate the lookup functions for types and values throughout the compiler.

In contrast, the compiler *does* produce the dynamic *imports* list. This turns out to be easy to do: the lambda-expression (intermediate representation) produced by the front end of the compiler explicitly contains these pids as free variables.

The dynamic pids suffice for type-safe linkage. Any inconsistency of static imports/exports that has no effect on dynamic imports/exports cannot cause execution to go wrong. [6]

6 Compilation interface

Previous "make" systems for SML/NJ accessed the compiler through the `use` procedure, which takes a file name, compiles the file, and side-effects the global environment by extending it with static/dynamic bindings.

[6] For example, IRM should recompile unit 2, below, if unit 1 changes to `type t=real`. But even if unit 2 were not recompiled, execution would not go wrong.

```
1. type t = int
2. local type u=t in val i=5 end
```

The primitives described in this paper are intended to be used with a *separate compilation manager* that understands source files, binary files, dates, directories, file-dependency DAGs, version, makefiles, and libraries. *IRM* [10], summarized in section 8, is such a system.

The SML/NJ compiler makes available an interface with one component for each phase of compilation: parsing, elaboration (semantic analysis) and hashing, code generation, and code execution. Also provided are several static-environment manipulation primitives: layering, index mapping, dehydration/rehydration, and so on.

The compiler itself does not know the format of a binary file. Instead, IRM takes the *compiledUnit*, dehydrates the *statenv*, and writes/reads the *statenv, code, imports, exports, import-static-pids, export-static-pid* to/from binary files in its own format.

The interactive top-level

Before the development of IRM and the mechanisms described in this paper, the interface to the SML/NJ compiler was an interactive read-eval-print loop with the provision for compiling some units from files (the `use` command). Of course, this provided no separate compilation whatsoever.

We have re-engineered the read-eval-print loop (and the `use` command) to call upon the compilation primitives described in this paper. Thus, there is only one interface into the compiler proper. Consequently, units compiled under IRM are compatible with units compiled in the top-level loop. (Of course, IRM cannot analyze the static dependences of source code typed in interactively.) And it should be easy (though we have not yet implemented it) for the interactive loop to load binary files without dragging in the full IRM compilation management system.

7 The Visible Compiler

The Lisp community has long enjoyed access to some internal functions of an interactive system (e.g. *eval*) using the "big ball of mud" paradigm. We show how to do this cleanly in a strongly typed, modular system, based on the model of compilation, linkage, and environments described here. We can selectively expose internal modules of the compiler simply by adding their static interfaces to the user-level environment.

Standard ML of New Jersey compiles ML code into machine code segments. The ML process can then execute these segments by putting them in memory (in the heap) and jumping to them. Linkage is achieved, as explained above, by looking up imported values in

a dynamic environment and passing these values as arguments to the machine code function.

In the interactive system, the user's program executes in the same process as the compiler itself. This is the usual way of building an interactive compilation environment (and would not be true of stand-alone ML applications).

Our compiler is, of course, compiled using the separate compilation system described in this paper. The compiled compiler consists of a set of binary files, each with machine code, static environment, pids, etc. To link the compiler, the runtime system has a primitive "bootstrap loader" (implemented in C) that understands only the machine code and import/export pids of the binary files. The bootstrap loader never reads the static environment component of the files. For each compilation unit in succession, the loader looks up values in a simple dynamic environment and applies machine code functions to them.

Once the compiler is running, it wants to make shared library modules (such as List, String, etc.), and perhaps its own internal modules, accessible to the user program in a type-safe way. To accomplish this, the ML function that initializes the environment for the interactive top-level loop re-reads the binary files, this time loading only the static environments and ignoring the machine code. These incremental static environments are rehydrated and composed to build a static environment matching the dynamic environment constructed by the bootstrap loader, which is passed as an argument to the ML startup function. The resulting combined environment contains all of the internal modules of the compiler, but it is normally filtered to provide only a limited selection of modules to be shared by the compiler and user programs.

Most of these shared modules are libraries providing built-in types or operating system interfaces, but others provide access to compiler control flags and to internal compiler types and operations, such as syntax trees, environments, and the compilation phases.

This "window" into the compiler has many applications:

- It supports shared fundamental libraries: List, String, I/O, etc.
- The separate compilation manager runs as an ordinary user program, but calls upon certain phases of the compiler. For example, it uses the parser to implement its own file dependency analysis.
- It's easy to implement a compiler hacker's testbed: one can add replacement modules to a running interactive compiler in which every module of the compiler is visible and usable.
- It provides a convenient basis for various forms of "metaprogramming", including the synthesis and evaluation of ML code by user programs.

8 IRM

Gene Rollins of Carnegie-Mellon University has implemented an *Incremental Recompilation Manager* (IRM) [15], which is a client of the low-level tools described in this paper. IRM runs as an ordinary "application program" in the SML/NJ interactive system, calling upon the compiler through the *Visible Compiler* interface.

We will only briefly summarize the capabilities of the IRM system. A user of IRM specifies groups of source files. A group may refer to other groups which it uses as libraries. The source files may be ML files or programs in other languages (Lex, Yacc, etc.) which are processed by compilers other than SML/NJ.

IRM automatically analyzes the dependencies between compilation units. For ML source files, it calls upon the parser of the SML/NJ compiler to scan files for free variables (imported names). For each file *foo.sml*, IRM keeps a cached version of the dependency information in *.d.foo.sml*, and uses timestamps to determine that the dependency information is up to date. However *intrinsic pids*—not timestamps—are used to ensure which files must be recompiled. IRM performs *cutoff* recompilation, not *cascading* recompilation.

IRM provides interfaces at several levels. The simplest—highest level—interface is a simple "makefile" listing the source files (by file name) and the names of imported libraries (by mentioning the name of the library's make file). Because dependencies are automatically computed by IRM, the makefile contains only an unordered list of file names. This system is simple and convenient.

A lower-level interface is via ML structures and signatures of components of the IRM system. Users with special needs (for example, a theorem prover whose "sources" are not kept in files; or a different style of library manager) can customize the IRM system, which has been implemented in a pleasingly modular style specifically to support such users.

9 Cross-compilation

The Visible Compiler requires the compiled code of the compiler to use the same representation for dehydrated environments as the compiler that produced it. The IRM system calls upon the same functions to compile top-level interactive commands (telling IRM to perform *make*, for example) as for generating binary files. These restrictions mean that cross-compilation (production of a new version of the compiler using an old version, or running the compiler on architecture A to generate code for architecture B) should be impossible.

A compiler unable to do these forms of cross-compilation is at an evolutionary dead end. We do support cross compilation, in the following ways. In generating code for a different target architecture, or for the same architecture with different parameter-passing conventions or garbage-collector interface (etc.) we do not use the "Visible Compiler" paradox. The "old" compiler loads all the binary files of the "new" compiler as a user program; only fundamental libraries such as buffered I/O are shared. The "old" compiler's phases are accessible to IRM as a standard structure *Compiler* with signature *COMPILER*.

The "new" compiler is loaded as a structure *NewCompiler* matching the same signature. The IRM system takes the compiler as a functor parameter; so it can be applied to *NewCompiler*. Now commands to IRM are compiled using the old compiler; but IRM itself calls upon *NewCompiler* to generate binary files. (Of course, the "old" interactive system should not attempt to execute these files!)

The many constraints between "old" and "new" compilers make cross compilation difficult to understand, but a simple axiomatization makes things much more manageable [2].

10 Alternatives

Dependencies between implementations arise from one implementation module mentioning another directly. If the interfaces were "fully expressive" (as they are in Modula 2) this would not matter; but in Standard ML, information about the identity of types can "leak through" the signatures—in other words, the signatures do not enforce abstraction. This is necessary, as explained in the introductory sections, to support the definition of higher-order modules with a single notion of signature matching.

Many of the difficulties which we have had to address in our work stem from these dependencies between implementation modules. Here we describe some alternative solutions that avoid these dependencies.

10.1 The fully-functorized style

The first solution is to adopt a programming style called "the fully functorized style" in which all external references are made opaque by making the external modules into functor parameters. This can be done in the existing Standard ML language; but in order to abstract over functors as well as structures, it's necessary to use higher-order functors, as described by Tofte and MacQueen [24, 18].

The problem with this approach is that it is very cumbersome for the programmer: the functor parameters must become complex enough to express the deeply nested dependency DAGs that should be handled automatically anyway. The DAGs must be explicitly and redundantly expressed in each implementation module.

10.2 Eliminate transparent signature matching

Another approach is to make all signature matching opaque, so that no information about the identity of types "leaks" through the interface (exported types are abstract). To make this practical, there must be a way to override this abstraction when we want to export a concrete type—but the overriding should be *explicit* in the signature, so that all dependencies are on interfaces.

Versions of the Standard ML module system with opaque signature matching have been proposed [11, 16]. But it appears that they are not true higher-order module systems, because dependencies that can be expressed at first order cannot be expressed at higher orders. The ML community is currently debating this issue. (See also Cardelli's proposed framework[6].)

A similar effect can be achieved in Standard ML by adding type abbreviations to signatures, and using "abstractions" (with opaque interfaces) instead of "structures" (with partially transparent interfaces). But this is even less powerful than the proposals mentioned in the previous paragraph.

11 Conclusion

The separate compilation mechanism described in this paper, with the IRM system implemented by Rollins[15], are regularly used by students in an undergraduate compiler course at Princeton. The combined IRM/Visible-Compiler system is robust enough that we rely on it to compile the SML/NJ compiler itself—a large system comprising over 200 compilation units.

The interfaces that express this compilation/linkage architecture are remarkably simple and easy to use. This is due to the functional style of environment manipulation, and the fact that executable-code linkage is expressed by function application.

The ML module language has long been recognized as powerful, but this work shows that it is also practical for organizing software systems. The fact that ML can support only *incremental recompilation* instead of true separate compilation is unfortunate, but we would not want to "solve" this problem by limiting power of parameterized modules, as is done in Modula-3. Instead, perhaps the research summarized in section 10.2 will eventually provide a better solution.

References

[1] Rolf Adams, Walter Tichy, and Annette Weiner. The cost of selective recompilation and environment processing. *ACM TOSEM*, 1994. to appear.

[2] Andrew W. Appel. Axiomatic bootstrapping: A guide for the compiler hacker. Technical Report CS-TR-451-94, Princeton University, March 1994.

[3] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–24, New York, 1987. Springer-Verlag.

[4] J. G. P. Barnes. *Programming in Ada plus Language Reference Manual*. Addison-Wesley, New York, 1991.

[5] Amber Benson and Gary Aitken. *OI Programmer's Guide*, June 1993.

[6] Luca Cardelli. Program fragments, linking, and modularization, April 1993.

[7] S. I. Feldman. Make – a program for maintaining computer programs. In *Unix Programmer's Manual, Seventh Edition, Volume 2A*. Bell Laboratories, 1979.

[8] Jurg Gutknecht. Separate compilation in modula-2: An approach to efficient symbol files. *IEEE Software*, pages 29–38, November 1986.

[9] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[10] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for standard ml of new jersey. Technical Report CMU-CS-94-116, Department of Computer Science, Carnegie-Mellon University, February 1994.

[11] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, pages 123–137, New York, Jan 1994. ACM Press.

[12] Bill Kalsow and Eric Muller. *SRC Modula-3 Version 1.6 manual*, February 1991.

[13] Andrew Koenig. Personal communication, 1993.

[14] Butler W. Lampson and Eric E. Schmidt. Organizing software in a distributed environment. In *ACM SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1–13, June 1983.

[15] Peter Lee, Gene Rollins, and Robert Harper. Incremental recompilation for Standard ML. Technical Report (in preparation), Carnegie-Mellon University, 1993.

[16] Xavier Leroy. Manifest types, modules, and separate compilation. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, pages 109–122, New York, Jan 1994. ACM Press.

[17] David B. MacQueen. The implementation of Standard ML modules. In *ACM Conf. on Lisp and Functional Programming*, pages 212–23, New York, 1988. ACM Press.

[18] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Proc. European Symposium on Programming (ESOP'94)*, page (to appear), April 1994.

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

[20] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[21] Mark Rain. Avoiding trickle-down recompilation in the Mary2 implementation. *Software—Practice and Experience*, 14(12):1149–1157, 1984.

[22] Ed Satterthwaite. e-mail, October 1993. Personal communication describing work done on the Cedar system at Xerox PARC in 1979–83 by Butler Lampson, Eric Schmidt, and Satterthwaite, but never published.

[23] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Proc. Twentieth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 439–450. ACM Press, 1993.

[24] Mads Tofte. Principal signatures for higher-order program modules. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 189–199, New York, Jan 1992. ACM Press.

[25] John von Neumann. First draft of a report on the EDVAC. In William Aspray and Arthur Burks, editors, *Papers of John von Neumann on Computing and Computer Theory*, pages 17–82. The MIT Press, Cambridge, Massachusetts, 1945.