Axiomatic Bootstrapping: A Guide for Compiler Hackers

ANDREW W. APPEL
Princeton University

If a compiler for language L is implemented in L, then it should be able to compile itself. But for systems used interactively commands are compiled and immediately executed, and these commands may invoke the compiler; so there is the question of how ever to cross-compile for another architecture. Also, where the compiler writes binary files of static type information that must then be read in by the bootstrapped interactive compiler, how can one ever change the format of digested type information in binary files?

Here I attempt an axiomatic clarification of the bootstrapping technique, using *Standard ML of New Jersey* as a case study. This should be useful to implementors of any self-applicable interactive compiler with nontrivial object-file and runtime-system compatibility problems.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; D.2.4 [**Software Engineering**]: Program Verification; D.2.6 [**Programming Environments**]: Interactive; D.4.9 [**Operating Systems**]: Systems Programs and Utilities—*linkers; loaders*

General Terms: Verification

Additional Key Words and Phrases: Bootstrapping

1. INTRODUCTION

A conventional C compiler, written in C, is said to be "bootstrapped" if it compiles itself. Now, suppose a new version of the compiler source is written, that uses different registers for passing arguments. The old compiler can compile this source, yielding a new compiler.

But look! The executable version `cc'` of the new compiler *uses the old* parameter-passing style, but *generates code* that uses the new style. One can use the new compiler, however, to recompile all the libraries (and the new compiler itself) and get a "new new" executable that both uses and generates the new parameter-passing style.

There is not much else to be said about bootstrapping C compilers (though see Section 6). But in a language with an interactive read-eval-print loop, commands are typed by the user, compiled immediately, and executed. Such a command, when compiled, may be a recursive call to the compiler itself, this time to compile a specified source file into a binary file. The compiler processing interactive commands

This work was supported in part by NSF Grant CCR-9200790. Author's address: Department of Computer Science, Princeton University, 35 Olden Street, Princeton NJ 08544-2087; e-mail: `appel@princeton.edu`.

Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

is the same one that compiles a source file; this makes it difficult to cross-compile for a different target architecture! In fact, in a sufficiently feature-laden interactive compilation system, there are many constraints on the retargeting and bootstrap process. This paper is a case study of the Standard ML of New Jersey (SML/NJ) system, explaining the difficulties and how to manage them.

Bin Files. Source files are transated by SML/NJ into "bin" files;[1] each bin file contains the executable machine code for the corresponding source, and the exported static environment for that source [Appel and MacQueen 1994]. For example, if a source file defines two structures S and T, each with several components, then the static part of the bin file is a description of the structures S and T: the names and types of their components and substructures.

The static part of the bin files is an ML data structure, complete with pointers, datatype constructors etc., created by the compiler and written in binary form to the bin file. Ordinarily, when a program is compiled to bin files by the interactive system, the bin files (including static part) will be read back into the same version of the system. But in bootstrapping, the compiled program is the "new" compiler, and we want to discard the "old" compiler. Thus, the new compiler executes, and loads in static information (from the bin files) about *itself.* For this to work, the representation of static environments in the old and new compilers must agree. This representation has two parts: the ML data types E (and their interpretation) chosen by the programmer; and the representation D of these data types as pointers and records in memory. Constraints on E might occur in any compiler that stores digested static information; constraints on D are a consequence of the fact that SML/NJ uses a *pickler* to write relocatable pointer data structures to the binary file just as they appear in memory.

In-line Primops. When the SML/NJ compiler first executes, it initializes its static environment by constructing a special *primitive basis* containing the in-line primitive operations (such as `+`, `:=`, etc.). This static environment is built using the same ML datatypes as ordinary static environments, but it is directly constructed without parsing any input. The primitive environment is then imported and used by the ML code that implements the initial basis.

When compiling a new version of the compiler, one can augment or change the primitive environment. But one cannot make use of the changed primitives until the new compiler compiles a "new new" compiler.

Initial Basis. ML programs can assume an "initial basis" [Milner et al. 1990, p. 77], an environment in which certain types and values are defined. The compiler itself also relies upon the initial basis. Furthermore, the source code for the initial basis is part of the source code for the compiler. Finally, the source code for the initial basis uses, in places, elements of the *primitive basis* containing names of specially implemented in-line functions.

This means that if some new version of the initial basis is desired, there are potential interactions with the rest of the system that must be considered.

Interactive System. The normal mode of operating SML/NJ is to compile both interactive input and ML programs from source files, *and run the compiled pro-*

[1]This is true of SML/NJ versions since 0.96.

grams in the same process as the compiler. It is particularly convenient to have an "interactive" compiler during the compiler development process, where individual compiler modules can be replaced "on-the-fly." The compiled programs use the same initial basis as the compiler. The compiled programs must be callable from the compiler, and must be able to call the same initial functions that the compiler calls; thus, compiled programs must use the same calling conventions (etc.) as the compiler itself. This makes it difficult to bootstrap a new version of the compiler that uses different calling conventions.

These interactions between the compiler and the executing program make for complications when (a new version of) the compiler *is* the executing program.

Example

The terminology of this example will be explained in Section 3, but the point here is to illustrate what can go wrong if one is not careful.

Suppose there is a version "1" of the system as a set of ML source files σ_1 and bin files β_1. The bin files are compiled object files (like "`.o`" files in a C system), and are the result of compiling the source files using some compatible version of the compiler.

We wish to use β_1 to compile σ_1, yielding a new version of the executable compiler.

First, `boot` builds an executable μ_1. This is like a link-loading step in a conventional system, but it must also load from β_1 the digested description of the initial static environment, so that compilations in μ_1 will have access to library modules (and the compiler itself) shared with μ_1. This sharing is essential, if only so that μ_1 and the interactive commands running within mu_1 do not have separate copies of runtime-system management data that would trip over each other.

$$\texttt{boot}(\beta_1) \;=\; \mu_1$$

Now μ_1 compiles the source σ_1 into binary object files β_1':

$$\texttt{compile}(\mu_1, \sigma_1) \;=\; \beta_1'$$

Now we hope that $\beta_1 \;\cong\; \beta_1'$, whatever that means.

However, suppose one edits the source files to produce σ_2, a new version of the compiler that uses a different calling convention. One might try the following steps:

$$\texttt{compile}(\mu_1, \sigma_2) \;=\; \beta_2$$

$$\texttt{boot}(\beta_2) \;=\; \bot$$

The `boot` step fails because β_2 is not self-consistent. The code generated by β_2 from a top-level interactive expression (using the new calling conventions) is able to call functions in the Basis within β_2 (compiled using the old conventions), so the first top-level command will dump core.

On the other hand, some changes are harmless: if the only change is a different algorithm for code optimization, `boot` will probably succeed. Finally, some self-inconsistencies will manifest themselves at stages other than `boot`.

How can one tell whether a change is harmless? And, since "nonharmless" changes are often necessary, how can one compile and bootstrap them correctly? The rest of the paper addresses these questions.

2. CHARACTERIZATION

I will use the following symbols to describe characteristics of a "version" of the compiler:

A. Architecture for which the compiler generates code, or on which it runs.

C. Calling conventions for which the compiler generates code: which registers are used for what purpose; how end-of-heap is detected; whether a stack is used; etc.

D. Datatype layout: how ML data types are laid out in memory.

E. Environment representation: how static environments are described in terms of ML data types.

B. Basis: the signature of the initial environment available to ordinary programs (and the compiler) upon startup.

P. Primitive basis: the static environment created by the compiler, describing in-line primitive operations and data types. Normally the primitive basis is used only in compiling the source code for the initial basis.

2.1 Source Characteristics

These characteristics are now used to describe the source code σ for some version of the compiler. The equations in this section just explain, in informal terms, the meaning of notation that will be used in the axioms of Section 3.

$a \in A_{\text{gen}}(\sigma)$. The compiler σ may contain code generators for several different target architectures; architecture a is a member of this set.

$C_{\text{gen}}(\sigma) = c$. The compiler σ generates code that uses the c calling conventions.

$D_{\text{gen}}(\sigma) = d$. The compiler σ generates code that uses the d datatype layout scheme.

$E_{\text{gen}}(\sigma) = e$. The compiler σ uses (and writes to bin files) the e static environment representation.

$B_{\text{use}}(\sigma) = b$. The nonbasis part of the system σ (that is, the compiler proper) is a program that makes use of functions in Basis b.

$B_{\text{imp}}(\sigma) = b$. The basis part of the system σ implements the basis b.

$P_{\text{use}}(\sigma) = p$. The basis part of the system σ is a program that makes use of the functions in primitive basis p.

$P_{\text{gen}}(\sigma) = p$. The compiler σ defines a primitive environment p for its compiled code.

2.2 Binary-File Characteristics

One can describe these aspects of compiled binary files in much the same way:

$A_{\text{run}}(\beta) = a$. The program β runs on architecture a.

$a \in A_{\text{gen}}(\beta)$. The compiler β generates code for the a architecture.

$C_{\text{run}}(\beta) = c$. The program β follows the c calling conventions.

$C_{\text{gen}}(\beta) = c$. The compiler β generates code that uses the c calling conventions.

$D_{\text{run}}(\beta) = d$. The internal data structures of program β obey the d datatype layout scheme.

$D_{\text{env}}(\beta) = d$. The static environments in bin files β use the d datatype layout scheme.

$D_{\text{gen}}(\beta) = d$. The compiler β generates code that uses the d datatype layout scheme.

$E_{\text{env}}(\beta) = e$. The static environment's bin files β are in the e environment representation.

$E_{\text{gen}}(\beta) = e$. The compiler β uses and generates the e environment representation.

$B_{\text{use}}(\beta) = b$. The nonbasis part of β (that is, the compiler proper) is a program that makes use of functions in basis b.

$B_{\text{imp}}(\beta) = b$. The basis part of β implements the basis b.

$P_{\text{gen}}(\beta) = p$. The compiler β_1 defines a primitive environment p_1 for its compiled code.

2.3 Executable-File Characteristics

The bin files are linked with a runtime system (and static environments are read from the bin files to initialize the compiler's user-visible "initial basis") to form an executable file μ, whose characteristics are just like those for bin files β, except that:

—Executable files do not have separate static environment sections as bin files do, so E_{env} and D_{env} do not apply.

—Executable systems generate code for only one machine, so $A_{\text{gen}}(\mu)$ is a single architecture rather than a set of architectures.

3. AXIOMS

I will give axioms describing the procedures of *compiling*, *bootstrapping*, *retargeting*, and *elaboration*; these axioms will then be used to prove theorems in Section 4.

3.1 Compiling

To compile source code, one executes the interactive system μ, and gives commands to compile source files σ into binary files β:

$$\texttt{compile}(\mu, \sigma) \;=\; \beta$$

for which the following equations must hold:

$$B_{\text{use}}(\sigma) \sqsubseteq B_{\text{imp}}(\sigma)$$
$$P_{\text{use}}(\sigma) \sqsubseteq P_{\text{gen}}(\mu)$$

The relation $\sqsubseteq$ expresses the ML signature-matching relation. That is, $P_{\text{use}}(\sigma) \sqsubseteq P_{\text{gen}}(\mu)$ means that the source files σ can be compiled in a primitive environment created by μ: every identifier looked up will be present and have an appropriate type. The "basis" (B) part of σ defines modules that are then used by the "compiler" part of σ, so the first equation is straightforward. But the "primitives" (P) containing special in-line function definitions must be specially constructed by μ.

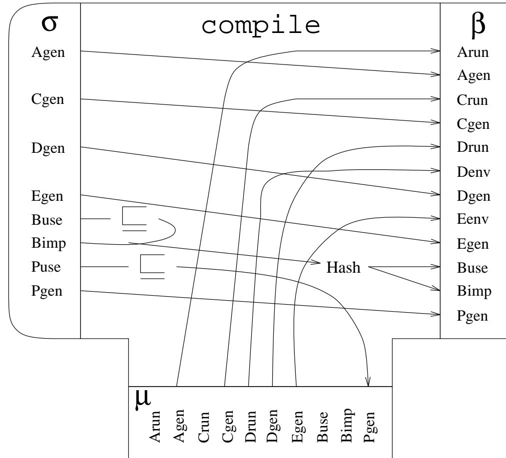

Fig. 1. "T-diagram" for `compile`.

The binary files (i.e., the files in the bin directory) β are then characterized by the following equations.

$$\begin{aligned}
A_{\text{run}}(\beta) &= A_{\text{gen}}(\mu) \\
A_{\text{gen}}(\beta) &= A_{\text{gen}}(\sigma) \\
C_{\text{run}}(\beta) &= C_{\text{gen}}(\mu) \\
C_{\text{gen}}(\beta) &= C_{\text{gen}}(\sigma) \\
D_{\text{run}}(\beta) &= D_{\text{gen}}(\mu) \\
D_{\text{gen}}(\beta) &= D_{\text{gen}}(\sigma)
\end{aligned}$$

These first six equations are unremarkable, and would occur in practically any compiler.

$$D_{\text{env}}(\beta) = D_{\text{run}}(\mu)$$

This equation results from the use of a "pickler" for writing the static type information (pointer data structures) to a file in the same binary format that is used in core.

$$\begin{aligned}
E_{\text{env}}(\beta) &= E_{\text{gen}}(\mu) \\
E_{\text{gen}}(\beta) &= E_{\text{gen}}(\sigma)
\end{aligned}$$

These two equations on E would hold in any compiler that writes digested static type information, with or without the use of a pickler.

$B_{\text{imp}}(\beta) = \mathcal{I}(B_{\text{imp}}(\sigma))$, where $\mathcal{I}$ is a hash function that computes "persistent identifiers" from the static environment exported by a source program. The persistent identifiers are then used for linking the machine code of different modules together, in a guaranteed type-safe way.

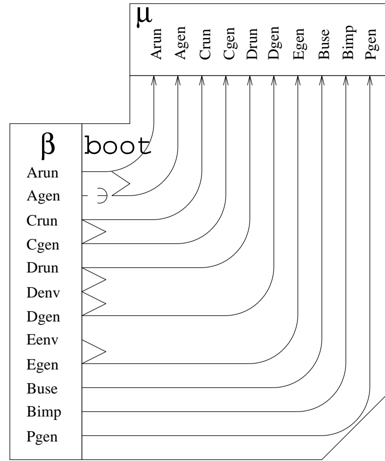

Fig. 2. Schematic for `boot`.

$B_{\rm use}(\beta) = B_{\rm imp}(\beta)$. The two equations on B are a consequence of sharing library code and data between the interactive system and the compiled user code.

$P_{\rm gen}(\beta) = P_{\rm gen}(\sigma)$. True in any compiler that defines functions that look ordinary to the user but are compiled specially (e.g., in-line).

These equations are summarized schematically in Figure 1.

3.2 Bootstrapping

The bootstrapper is a part of the C language runtime system. It knows just enough to extract the dynamic part (machine code) from bin files β; but not the format of static environment representations, which only the compiler understands. However, the machine code within β *is* the compiler; once it starts running, it can load the static part of β to form an environment (symbol table of the initial basis) for compiling user programs. The result is an interactive compiler μ:

$$\texttt{boot}(\beta) \ = \ \mu.$$

For this to work, the following equations must hold:

$A_{\rm run}(\beta) \in A_{\rm gen}(\beta)$. So that the compiler and top-level interactive commands can both run on the same computer.

$C_{\rm run}(\beta) = C_{\rm gen}(\beta)$. So that top-level interactive commands can call and be called by the compiler and initial basis.

$D_{\rm run}(\beta) = D_{\rm gen}(\beta)$. for the same reason.

$D_{\rm env}(\beta) = D_{\rm run}(\beta)$. So the bootstrapping compiler can read static environments from bin files.

$E_{\rm env}(\beta) = E_{\rm gen}(\beta)$. For the same reason.

The remaining equations characterize the output μ:

$$A_{\rm run}(\beta) \ = \ A_{\rm run}(\mu) = A_{\rm gen}(\mu)$$
$$C_{\rm run}(\beta) \ = \ C_{\rm run}(\mu)$$

$$\begin{aligned} C_{\text{gen}}(\beta) &= C_{\text{gen}}(\mu) \\ D_{\text{run}}(\beta) &= D_{\text{run}}(\mu) \\ D_{\text{gen}}(\beta) &= D_{\text{gen}}(\mu) \\ E_{\text{gen}}(\beta) &= E_{\text{gen}}(\mu) \\ B_{\text{use}}(\beta) &= B_{\text{use}}(\mu) \\ B_{\text{imp}}(\beta) &= B_{\text{imp}}(\mu) \\ P_{\text{gen}}(\beta) &= P_{\text{gen}}(\mu) \end{aligned}$$

These equations are summarized in Figure 2.

Now, for example, one can see that the `boot` failure described in Section 1 is because $C_{\text{run}}(\beta_2) \neq C_{\text{gen}}(\beta_2)$ violating one of the preconditions for `boot`.

3.3 Retargeting

Because it is impossible to bootstrap using `compile` and `boot` if the new compiler uses a new calling sequence or environment representation, two special procedures are provided. The first of these is called `retarget`: Run an interactive compiler μ_1, and load the bin files β for a *different* version of the compiler *as a "user program."* Since β may include code generators for many machines, one can also specify which target architecture a's code generator should be selected from β.

$$\texttt{retarget}(\mu_1, \beta, a) = \mu_2$$

The compiler originally present in μ_1 will be used in μ_2 for compiling top-level interactive commands, but the compiler β will be used in μ_2 for turning source files into bin files.

Ordinary user programs do not provide their own implementation of the initial basis, so the basis portion of β (corresponding to the that implement $B_{\text{imp}}(\beta)$) will not be loaded: $B_{\text{imp}}(\beta)$ is irrelevant. However, the nonbasis portion of the compiler β must be compatible with the basis already running in μ_1, so that β can call upon standard I/O functions (etc.) built into μ_1.

As an ordinary user program, the bin files β executing under the supervision of μ_1 can generate code for any architecture or any calling sequence. This is because the code is not going to be executed in the current process, so it need not be compatible with the instruction set or calling conventions that μ_1 itself is using. This freedom is crucial for cross-compilation (compilation for a different target architecture or calling convention).

The following restrictions apply (see Figure 3):

$$\begin{aligned} a &\in A_{\text{gen}}(\beta) \\ A_{\text{run}}(\beta) &= A_{\text{run}}(\mu_1) \\ C_{\text{run}}(\beta) &= C_{\text{run}}(\mu_1) \\ D_{\text{run}}(\beta) &= D_{\text{env}}(\beta) = D_{\text{run}}(\mu_1) \\ E_{\text{env}}(\beta) &= E_{\text{gen}}(\mu_1) \\ B_{\text{use}}(\beta) &= B_{\text{imp}}(\mu_1) \end{aligned}$$

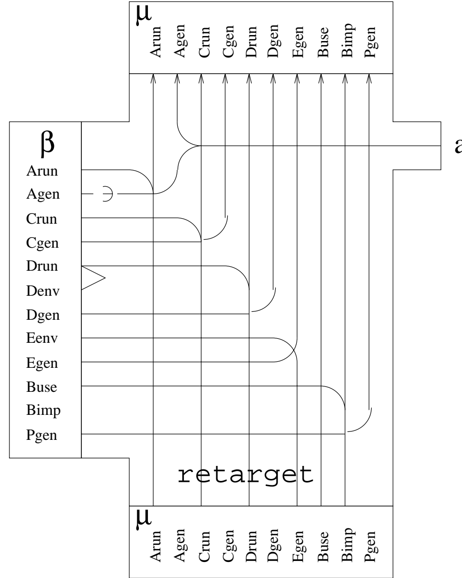

Fig. 3. Schematic for `retarget`.

The last equation is an *exact* signature match. In particular, it means that the intrinsic persistent identifiers generated from the compilation of the initial basis (files in the `src/boot` directory) in building the bin files within μ_1 must be identical to the corresponding identifiers in the initial basis portion of β. [2]

The following equations characterize the output μ_2:

$$\begin{aligned} A_{\text{run}}(\mu_2) &= A_{\text{run}}(\mu_1) \\ A_{\text{gen}}(\mu_2) &= a \\ C_{\text{run}}(\mu_2) &= C_{\text{run}}(\mu_1) \\ C_{\text{gen}}(\mu_2) &= C_{\text{gen}}(\beta) \end{aligned}$$

[2]This can be guaranteed by producing β and μ_1 from the same compiler μ_0, in the following way:

$$\begin{aligned} \texttt{compile}(\mu_0, \sigma_1) &= \beta_1 \\ \texttt{boot}(\beta_1) &= \mu_1 \\ \texttt{compile}(\mu_0, \sigma_2) &= \beta \end{aligned}$$

where the source codes for the B_{imp} portions of σ_1 and σ_2 are *identical*.

This works because $\mathcal{I}$ is really a function: $(x = y) \Rightarrow (\mathcal{I}(x) = \mathcal{I}(y))$.

In versions 0.96–0.98 of the SML/NJ system, the "persistent identifiers" were just timestamps, so that $\mathcal{I}$ would return different results at different times. Therefore, with the procedure outlined at the top of this footnote, $B(\beta)$ and $B(\beta_1)$ would not export the same "persistent identifiers" even though the source code was identical. Instead, one would have to create a new, empty bin directory; copy *just the bin files for the initial basis* from β_1 to the new bin directory β; and then proceed with a `compile` that would use these files as a starting point.

$$\begin{aligned}
D_{\text{run}}(\mu_2) &= D_{\text{run}}(\mu_1) \\
D_{\text{gen}}(\mu_2) &= D_{\text{gen}}(\beta) \\
E_{\text{gen}}(\mu_2) &= E_{\text{gen}}(\beta) \\
B_{\text{use}}(\mu_2) &= B_{\text{use}}(\mu_1) \\
B_{\text{imp}}(\mu_2) &= B_{\text{imp}}(\mu_1) \\
P_{\text{gen}}(\mu_2) &= P_{\text{gen}}(\beta)
\end{aligned}$$

This is funny hybrid indeed.

3.4 Elaboration

Finally, `elab` is a special variation on `boot` that reparses the source files to build the static environment, instead of reading it from the bin files: $\texttt{elab}(\beta, \sigma) = \mu$.

Now, given the two steps

$$\begin{aligned}
\texttt{compile}(\mu_0, \sigma) &= \beta \\
\texttt{elab}(\beta, \sigma) &= \mu
\end{aligned}$$

β must satisfy all the equations given for `boot` above *except* for the ones involving $D_{\text{env}}(\beta)$ and $E_{\text{env}}(\beta)$, because the environments will not be read from the bin files.

Another requirement for `elab` is that σ and β must be related by the `compile` command shown (see Figure 4).

The resulting executable μ is defined by the same equations as for $\texttt{boot}(\beta)$. In fact, with `elab` there is no need for `boot`, except that `elab` is much slower because it reparses all the source.

4. STABLE VERSIONS

Definition. (σ, β) form a *stable version* if the following equations hold:

$$A_{\text{run}}(\beta) \in A_{\text{gen}}(\sigma) = A_{\text{gen}}(\beta)$$

$$C_{\text{gen}}(\sigma) = C_{\text{run}}(\beta) = C_{\text{gen}}(\beta)$$

$$D_{\text{gen}}(\sigma) = D_{\text{run}}(\beta) = D_{\text{gen}}(\beta) = D_{\text{env}}(\beta)$$

$$E_{\text{gen}}(\sigma) = E_{\text{env}}(\beta) = E_{\text{gen}}(\beta)$$

$$B_{\text{use}}(\sigma) \sqsubseteq B_{\text{imp}}(\sigma)$$

$$B_{\text{imp}}(\beta) = \mathcal{I}(B_{\text{imp}}(\sigma))$$

$$B_{\text{use}}(\beta) = B_{\text{imp}}(\beta)$$

$$P_{\text{use}}(\sigma) \sqsubseteq P_{\text{gen}}(\sigma) = P_{\text{gen}}(\beta)$$

Remark: If $\texttt{compile}(\texttt{boot}(\sigma, \beta'), \sigma) = \beta'$ then (σ, β) is a *fixed point*, a stronger property. But we cannot prove fixed-point properties from the axioms in this paper.

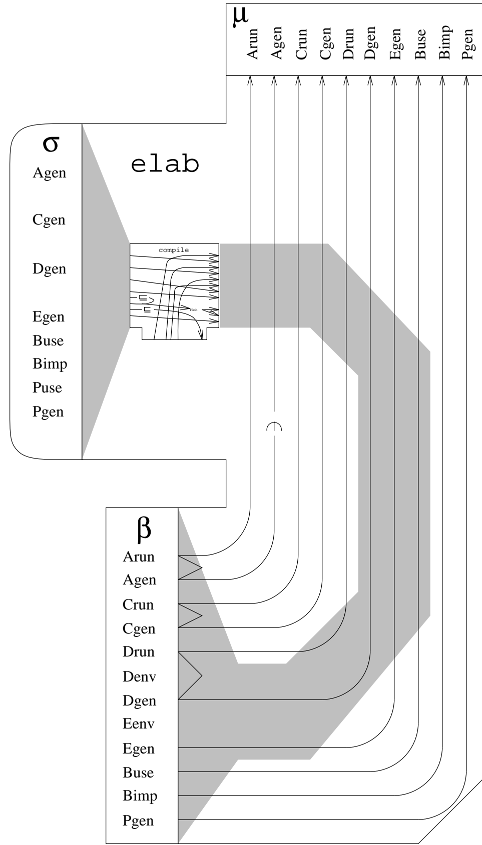

Fig. 4. Schematic for `elab`.

Suppose one starts with a stable version (σ_1, β_1) and creates a new source σ_2. How can one obtain bin files β_2 to make a stable version with the new source?

All of the "theorems" in this section are proved using only the axioms of Section 3 and the definition of a stable version.

4.1 New Primitive Basis

Suppose $P_{\text{gen}}(\sigma_2) \neq P_{\text{gen}}(\sigma_1)$, $P_{\text{use}}(\sigma_2) \sqsubseteq P_{\text{gen}}(\sigma_2)$, but all other characteristics (A_{gen}, C_{gen}, D_{gen}, E_{gen}, B_{use}, B_{gen}, P_{use}) are identical from one source to another. Then

$$\begin{aligned} \texttt{boot}(\beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2. \end{aligned}$$

The reader can verify that both of these steps succeed, and that (σ_2, β_2) is stable.

4.2 New Initial Basis

Now suppose σ_2 differs from σ_1 in the initial basis (e.g., standard library, which is implemented in and used by the compiler, and is also used by client programs); and perhaps also in P_{use}, the set of primitives used in the initial basis.

$$\begin{aligned} \texttt{boot}(\beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \end{aligned}$$

The reader can verify that these steps succeed and result in (σ_1, β_2) stable.

4.3 New Environments

Suppose σ_2 uses a different environment representation (E_{gen}) from σ_1.
The "ordinary" procedure will not work:

$$\begin{aligned} \texttt{boot}(\beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \\ \texttt{boot}(\beta_2) &= \bot \end{aligned}$$

Now $E_{\text{gen}}(\beta_2) \neq E_{\text{env}}(\beta_2)$, so β_2 cannot be used in `boot`. There are two ways to build a stable version:

$$\begin{aligned} \texttt{elab}(\beta_2, \sigma_2) = \mu_2 \\ \texttt{compile}(\mu_2, \sigma_2) = \beta_2' \end{aligned} \qquad \text{or} \qquad \begin{aligned} \texttt{retarget}(\mu_1, \beta_2, A_{\text{run}}(\mu_1)) = \mu_2' \\ \texttt{compile}(\mu_2', \sigma_2) = \beta_2'' \end{aligned}$$

Now, (σ_2, β_2') is stable, and so is (σ_2, β_2''); β_2' and β_2'' are equivalent in all properties.

4.4 New Datatype Layout

If the new compiler uses a different datatype layout (that is, $D_{\text{gen}}(\sigma_2) \neq D_{\text{gen}}(\sigma_1)$) then the following steps will build a stable version.

$$\begin{aligned} \texttt{boot}(\beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \\ \texttt{elab}(\beta_2, \sigma_2) &= \mu_2 \end{aligned}$$

`Retarget` will not do the job; for suppose

$$\begin{aligned} \texttt{boot}(\beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \\ \texttt{retarget}(\mu_1, \beta_2, a) &= \mu_2' \\ \texttt{compile}(\mu_2', \sigma_2) &= \beta_2' \end{aligned}$$

then $D_{\text{env}}(\beta_2') = D_{\text{gen}}(\sigma_1)$, while $D_{\text{run}}(\beta_2') = D_{\text{gen}}(\sigma_2)$. Thus β_2' cannot be used as input to either `boot` or `retarget`.

4.5 New Calling Conventions

Suppose σ_2 uses new calling conventions: $C_{\text{gen}}(\sigma_2) \neq C_{\text{gen}}(\sigma_1)$.

The procedure is:

$$\begin{aligned}
\texttt{boot}(\beta_1) &= \mu_1 \\
\texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \\
\texttt{retarget}(\mu_1, \beta_2, A_{\text{run}}(\mu_1)) &= \mu_2 \\
\texttt{compile}(\mu_1', \sigma_2) &= \beta_2'
\end{aligned}$$

Now (σ_2, β_2') is stable. The reader can verify that `retarget` is necessary, and that `elab` would not suffice.

4.6 New Target Architecture

Given (σ, β) stable, $A_{\text{run}}(\beta) = a_1$, suppose one wishes to make a compiler that runs on architecture a_2, for $a_2 \in A_{\text{gen}}(\beta)$.

$$\begin{aligned}
\texttt{boot}(\beta) &= \mu_1 \\
\texttt{retarget}(\mu_1, \beta, a_2) &= \mu_2 \\
\texttt{compile}(\mu_2, \sigma) &= \beta_2
\end{aligned}$$

Now (σ, β_2) is a stable compiler running on, and generating code for, architecture a_2.

4.7 Getting from Here to There

Suppose there is a stable version (σ_1, β_1), and a compiler $\mu_1 = \texttt{boot}(\beta_1)$. The programmer makes a new source σ_2 that differs in every characteristic from σ_1. Let us assume, however, that $P_{\text{use}}(\sigma_1) \sqsubseteq P_{\text{gen}}(\sigma_2)$.

There may well exist a β_2 such that (σ_2, β_2) is stable, but we do not have such a β_2. How is it to be obtained?

The first problem is that `compile` is inapplicable, since we cannot assume either $B_{\text{use}}(\sigma_2) \sqsubseteq B_{\text{imp}}(\sigma_2)$ or $P_{\text{use}}(\sigma_2) \sqsubseteq P_{\text{gen}}(\mu_1)$.

The procedures `boot`, `retarget`, and `elab` are not useful, since they just take the binary files β_1 that we already have. $\texttt{elab}(\beta_1, \sigma_2)$ is illegal (as the reader may verify), and the author cannot even imagine why it might be useful.

The trick is to make some intermediate versions of the source code: σ_x is like σ_1 but defines augmented primitives P; σ_y is like σ_x, but makes use of the augmented primitives and provides an augmented basis B_{imp}.

So, σ_x is as follows:

$$\begin{aligned}
A_{\text{gen}}(\sigma_x) &= A_{\text{gen}}(\sigma_1) \\
C_{\text{gen}}(\sigma_x) &= C_{\text{gen}}(\sigma_1) \\
D_{\text{gen}}(\sigma_x) &= D_{\text{gen}}(\sigma_1) \\
E_{\text{gen}}(\sigma_x) &= E_{\text{gen}}(\sigma_1)
\end{aligned}$$

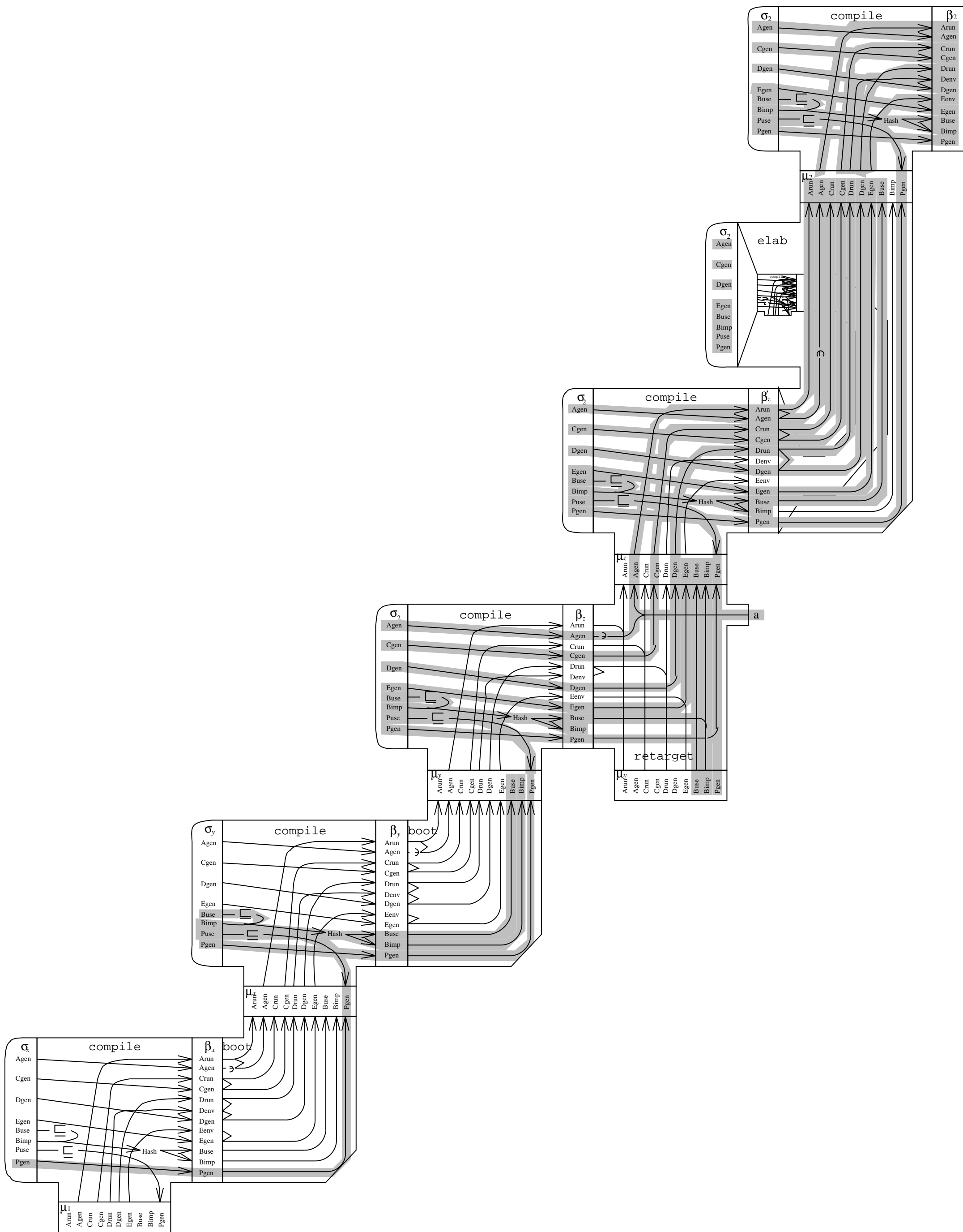

Fig. 5. Getting from here to there.

$$
\begin{aligned}
B_{\text{use}}(\sigma_x) &= B_{\text{use}}(\sigma_1)\\
B_{\text{imp}}(\sigma_x) &= B_{\text{imp}}(\sigma_1)\\
P_{\text{use}}(\sigma_x) &= P_{\text{use}}(\sigma_1)\\
P_{\text{gen}}(\sigma_x) &= P_{\text{gen}}(\sigma_2)
\end{aligned}
$$

Then:

$$
\begin{aligned}
\texttt{compile}(\mu_1, \sigma_x) &= \beta_x\\
\texttt{boot}(\beta_x) &= \mu_x
\end{aligned}
$$

Now version σ_y is another intermediate version:

$$
\begin{aligned}
A_{\text{gen}}(\sigma_y) &= A_{\text{gen}}(\sigma_1)\\
C_{\text{gen}}(\sigma_y) &= C_{\text{gen}}(\sigma_1)\\
D_{\text{gen}}(\sigma_y) &= D_{\text{gen}}(\sigma_1)\\
E_{\text{gen}}(\sigma_y) &= E_{\text{gen}}(\sigma_1)\\
B_{\text{use}}(\sigma_y) &= B_{\text{use}}(\sigma_2)\\
B_{\text{imp}}(\sigma_y) &= B_{\text{imp}}(\sigma_2)\\
P_{\text{use}}(\sigma_y) &= P_{\text{use}}(\sigma_2)\\
P_{\text{gen}}(\sigma_y) &= P_{\text{gen}}(\sigma_2)
\end{aligned}
$$

Now:

$$
\begin{aligned}
\texttt{compile}(\mu_x, \sigma_y) &= \beta_y\\
\texttt{boot}(\beta_y) &= \mu_y\\
\texttt{compile}(\mu_y, \sigma_2) &= \beta_z\\
\texttt{retarget}(\mu_y, \beta_z) &= \mu_z\\
\texttt{compile}(\mu_z, \sigma_2) &= \beta'_z\\
\texttt{elab}(\beta'_z, \sigma_2) &= \mu_2\\
\texttt{compile}(\mu_2, \sigma_2) &= \beta_2
\end{aligned}
$$

Now (σ_2, β_2) is stable. The proof is just simple (but tedious) equational reasoning, checking that the preconditions of each step are satisfied and characterizing the intermediate results $\beta_x, \mu_x, \beta_y, \mu_y$, etc.

Figure 5 presents this proof schematically, where grey lines indicate the σ_2 version of each characteristic.

It does seem amazing that five compilations are required to get from stable version 1 to stable version 2. But I have not found a shorter sequence.

5. GENERALITY

In what sense do the "characteristics" A, C, D, E, B, P form, in any sense, a complete set?

The axioms cannot assure the correctness of the compiler. Specifying that a 50,000-line program implements faithfully the 100-page *Definition of Standard ML*[Milner et al. 1990] is not something that can be done with eight or ten simple equations in the style shown in this paper. The axioms are meant as abstractions of only those aspects of bootstrapping that often prove problematical. Many other aspects of ML compilation, though difficult or interesting, pose no special problems when bootstrapping and are entirely ignored by the axioms.

However, perhaps there are other important issues related to bootstrapping that are not accurately characterized by any of the axioms.

5.1 Runtime System

ML requires a runtime system, to do garbage collection, to handle system calls, and to provide various functions implemented in C or assembly language. The runtime system must know the format of ML data types (to do garbage collection) and must satisfy other constraints. A runtime system ρ has the properties $A_{\text{run}}(\rho)$, the architecture on which it runs; $C_{\text{run}}(\rho)$, the calling conventions for ML-callable entry points; and $D_{\text{run}}(\rho)$, the ML datatype layout that it understands.

To model runtime systems, we extend `boot` with a runtime-system argument: $\texttt{boot}(\rho, \beta) = \mu$ with extra preconditions

$$\begin{aligned} A_{\text{run}}(\beta) &= A_{\text{run}}(\rho) \\ C_{\text{run}}(\beta) &= C_{\text{run}}(\rho) \\ D_{\text{run}}(\beta) &= D_{\text{run}}(\rho) \end{aligned}$$

Elaboration also requires a particular runtime system: $\texttt{elab}(\rho, \beta, \sigma)$ with the same three preconditions.

The implications of these constraints turn out to be quite trivial; runtime system issues cause no bootstrapping problems, except as described in the next subsection.

5.2 Structured I/O Format

For example, John Reppy recently rewrote the "pickler" in the runtime system, that writes pointer data structures to files (and reads them back). In particular, the pickler writes static environment representations to bin files β. Reppy's new pickler uses a different file format from the old one. The implementation of (either version of) the pickler, and any knowledge about file format, is entirely within the runtime system.

We could characterize this as $F_{\text{gen}}(\rho)$, the format that a given runtime system uses to write ML datatypes to a file. Then the bin file β would have a characteristic $F_{\text{env}}(\beta)$, the format in which static environments have been written; and executables μ would have the characteristic $F_{\text{gen}}(\mu)$ based on the format that μ's runtime system uses.

Then we have the following additional axioms. For $\texttt{compile}(\mu, \sigma) = \beta$ we have $F_{\text{gen}}(\mu) = F_{\text{env}}(\beta)$.

For $\texttt{boot}(\rho, \beta) = \mu$ we have

$$\begin{aligned} F_{\text{env}}(\beta) &= F_{\text{gen}}(\rho) \\ F_{\text{gen}}(\mu) &= F_{\text{gen}}(\rho) \end{aligned}$$

(the first is a precondition, the second characterizes the output μ).

For `retarget`$(\mu_1, \beta, a) = \mu_2$ we have

$$F_{\text{env}}(\beta) = F_{\text{gen}}(\mu_1)$$
$$F_{\text{gen}}(\mu_2) = F_{\text{gen}}(\mu_1).$$

And finally, for `elab`$(\rho, \beta, \sigma) = \mu$ we have only $F_{\text{gen}}(\mu) = F_{\text{gen}}(\rho)$, and $F_{\text{env}}(\beta)$ irrelevant.

Clearly, Reppy will need to use `elab` in order to bootstrap his new structure-blaster format, since `boot` and `retarget` are too restrictive.

This example has illustrated that the axioms of Section 3 do not necessarily form a complete set, but the axiomatic method is easily extensible to meet new challenges.

5.3 New Module-Field Layout

Older versions of SML/NJ sorted the value fields of a signature into alphabetical order before generating code. This meant that the translation of this module S

```
structure S =
struct
 val b = 5
 val a = 7
end
```

would be as a record in memory in which a (7) appeared first, followed by b (5).

Current versions of SML/NJ do not sort into alphabetical order. Thus, bin files compiled by the new version should be incompatible with executables of the old version.

Consider the axiomatization. We say that:

$G_{\text{gen}}(\sigma)$ is the sorting (nonsorting) technique used for structure fields by source code σ;

$G_{\text{run}}(\beta)$ is the structure-field layout algorithm that had been used in compiling β;

$G_{\text{gen}}(\beta_1)$ is the structure-field layout algorithm that β uses in generating output code;

$G_{\text{run}}(\mu)$ is analogous to $G_{\text{run}}(\beta)$;

$G_{\text{gen}}(\mu)$ is analogous to $G_{\text{gen}}(\beta)$;

$G_{\text{run}}(\rho)$ is the ordering that ρ uses for interfacing its own "primitive" structures visible from the ML program.

The next step is to write axioms for G. This is not trivial, as it involves an understanding of how the compiler and generated code work. It turns out, however, that the equations for G in the steps `compile`, `boot`, `retarget`, `elab` are exactly parallel to the equations for C. This implies that G was not necessary at all, and that C expresses (among other things) the ordering of structure fields. This is a measure of the robustness of the original axioms.

5.4 Record Field Ordering

Record fields are also sorted by label in SML/NJ. Sorting is required by the semantics of the language, but any consistent ordering will do. The sorting could, in

principle, be done in some nonstandard (i.e. nonalphabetical) order.

Since records are used directly in the implementation of static environments, and structures are not, the effect of record fields turns out to be axiomatized exactly like datatype layouts D, not like calling sequences C. This should not be surprising, as the record type `{a,b}` is indeed a kind of type constructor (just like a datatype) and the layout into bits of ML data types is exactly what D was supposed to characterize.

6. RELATED WORK

Lecarme et al. [1982] present a good explanation of a theory of bootstrapping using T-diagrams, a notation invented by Bratman [1961] and formalized by Earley and Sturgis [1970]. This theory is simple and elegant, and the diagrams are pretty to look at. It is very successful in describing the steps needed to produce a compiler from source language SL to object language OL written in implementation language WL, when one has (for example) a machine executing instruction set XL, a translator from WL to XL implemented in XL, an interpreter for OL written in AL, and a translator for AL written in ..., and so on. The T-diagrams seem more compact, and easier to read once one learns how, than the corresponding equational theory.

In fact, Earley and Sturgis provide an algorithm to construct a bootstrap sequence: given a set of translators and interpreters (characterized by source, object, and implementation languages), and a desired translator (similarly characterized) their algorithm can either show how to construct the desired result or prove that it cannot be done. Perhaps an algorithm such as this could be devised to prove the theorems of Sections 4.1–4.6.

Lecarme goes further, with a flowchart that provides hints about what existing translator should be modified "by hand" (to produce a different target language, or to accept a different source language, or to run in a different implementation language) to get to the desired result. Note the similarities with the hand-made intermediate versions σ_x, σ_y needed in Section 4.7.

The added problem in SML/NJ (and in similar interactive systems, especially those that have predigested type information) is that there are extra constraints between the implementation language and the object language that "opaque" T-diagrams do not express. Furthermore, the different languages in question are all quite similar: executable code described by (in this case) six characteristics, where many of the characteristics are likely to match between any two versions. In using opaque T-diagrams, the similarities between two executables (e.g., identical data type representation) are lost, and would have to be expressed separately in a set of equations. This paper has demonstrated T-diagrams with internal structure (representing equational constraints), which are more powerful than "opaque" T-diagrams.

Thompson's Turing award lecture [Thompson 1984] describes from a different point of view how bugs (and viruses) can propagate through the bootstrapping process.

7. CONCLUSION

When compiled code shares important parts of the environment with the compiler itself, bootstrapping new versions can be complicated, and previous theories of

bootstrapping do not seem to extend well. Clearly written axioms can help the poor compiler hacker deal with the complexity.

Certain choices made in the SML/NJ system complicate bootstrap process:

—The SML/NJ system uses a "pickler" to write static environments to binary files in (almost) exactly the same format that's used in memory. The resulting constraints on $D_{\mathrm{env}}(\beta)$ cause the procedures of Sections 4.4 and 4.7 to take extra `elab` steps.

—The sharing of code and data between the compiler and user programs requires the compiler to load its own static environments, causing constraints on E and B.

—The use of the same compiler for compiling interactive commands and source files for the compiler itself requires a special `retarget` mechanism for relaxing constraints on A and C.

However, each of these features is useful in its own way. The axiomatization of bootstrapping makes it easier to tolerate complexity in the process, so that these features can be more easily supported.

Appendix: Command realization

Each of the abstract functions `compile`, `boot`, `retarget`, `elab` corresponds to a sequence of operating-system commands (a shell script) or ML commands. Let μ be an interactive `sml` executable with the *Compilation Manager* (**make** system) loaded, called `sml-cm`. Let σ be a set of source files for the compiler in directory `src`. Then `compile`(μ, σ) is just

```
cd src; echo "Batch.make()" | sml-cm
```

Supposing that the target architecture is `sparc` ($A_{\mathrm{gen}}(\mu) =$ `sparc`), this creates a directory $\beta =$ `bin.sparc` containing bin files.

Bootstrapping (`boot`(ρ, β)) is done by two shell scripts: `makeml` compiles the runtime system ρ (written in C and assembly language) from the `src/runtime` subdirectory, runs it to load the bin files β to create an executable `sml`, and `makecm` executes `sml` to load the Compilation Manager, creating an executable $\mu =$`sml-cm`:

```
cd src; makeml -bin bin.sparc; makecm
```

Retargeting is done by instructing the compilation manager $\mu =$ `sml-cm` to load bin files for an alternate compiler in directory $\beta =$ `alt/bin.sparc` (for example) and to select the `alpha` code generator within those files:

```
echo 'retarget("alt/bin.sparc",".alpha"); exportML("sml-a")' | sml-cm
```

The result of this `retarget`(μ, β, α) is $\mu_2 =$`sml-a`.

Finally, elaboration is just like `boot` but with an extra command-line flag `-elab` to `makeml`.

REFERENCES

Appel, A. W. and MacQueen, D. B. 1994. Separate compilation for Standard ML. In *Proceedings of SIGPLAN '94 Symposium on Programming Language Design and Implementation.* ACM Press, New York, 13–23.

BRATMAN, H. 1961. An alternate form of the UNCOL diagram. *Commun. ACM 4,* 3 (Mar.), 142.

EARLEY, J. AND STURGIS, H. 1970. A formalism for translator interactions. *Commun. ACM 13,* 10 (Oct.), 607–617.

LECARME, O., PELLISSIER, M., AND THOMAS, M.-C. 1982. Computer-aided production of language implementation systems: A review and classification. *Softw. Pract. Exp. 12,* 9, 785–824.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Mass.

THOMPSON, K. 1984. Reflections on trusting trust. *Commun. ACM 27,* 8 (Aug.), 761–763.

Received January 1994; revised March 1994; accepted March 1994.