Axiomatic Bootstrapping: A guide for compiler hackers

Andrew W. Appel
CS-TR-451-94
Princeton University
appel@princeton.edu

March 10, 1994

Abstract

The Standard ML of New Jersey compiler is written in Standard ML. Thus, SML/NJ can compile itself. This should be simple and obvious.

There are some complications. The compiler produces binary files that contain both *executable code* and *static environments* (mapping identifiers to structures, signatures, and types). Normal binary files are intended to be loaded into a running SML/NJ system, where the static environment enhances the environment already available in that system, and the code is loaded and run.

Some new version of the compiler may make different assumptions about the format of static environments, and its executing code may not be compatible with the existing interactive system. Thus, there can be a severe "bootstrap" problem: how can the old version of the compiler compile and bootstrap the new version?

Here I attempt an axiomatic clarification of the bootstrapping technique. This should be useful to implementors of any self-applicable compiler with nontrivial object-file and runtime-system compatibility problems.

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 re-compile 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's not much else to be said about bootstrapping C compilers.[1] Why, then, is it worth a whole paper to talk about bootstrapping the Standard ML of New Jersey (SML/NJ) compiler?

Bin files: Source files are transated by SML/NJ into "bin" files;[2] each bin file contains the executable machine code for the corresponding source, and the exported static environment for that source.[1] 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. When an ordinary program (other than the compiler) is compiled to bin files, the bin files (including static part) will be read back into the same version of the compiler that created it. But in bootstrapping, the dynamic part of the bin file (executable machine code) will execute, and then read in its own static part. Thus, the representation of static environments in the old and new compilers must agree. There are two parts to this representation: The translation of ML static semantics into ML data types, and the translation of ML data types into binary pointer and record layout.

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

[1] This is an exaggeration; see section 7.
[2] This is true of SML/NJ versions since 0.96.

itive environment is then imported and used by the ML code that implements the initial basis ("pervasives").

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

Pervasive Basis: ML programs can assume an "initial basis" [5, page 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 in-line primops.

This means that if some new version of the initial basis is desired, there are potential interactions with the rest of the compiler 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 programs in the same process as the compiler.* The compiled programs use the same pervasive basis as the compiler. The compiled programs must be callable from the compiler, and must be able to call the same pervasive 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 compiler as a set of ML source files σ_1, runtime system ρ_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, `makeml` builds an `sml-scb` executable μ_1 (this is like a link-loading step in a conventional compiler, but it must also load from β_1 the digested description of the initial static environment):

$$\texttt{makeml}(\rho_1, \beta_1) \;=\; \mu_1$$

The first argument to `makeml` is the source code, used only for its `runtime` subdirectory containing the C code for the runtime system.

Now the `sml-scb` executable (μ_1) compiles the source σ_1 into binary object files β_1':

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

Now it should be 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:

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

$$\texttt{makeml}(\rho_1, \beta_2) \;=\; \bot$$

The `makeml` step fails because β_2 is not self-consistent. The code generated by β_2 when compiling a top-level interactive expression will not be compatible with the Basis (which was compiled by β_1), so the first top-level command (`ExportML("sml")`) executed by the `makeml` script will dump core.

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

How can one tell whether a change is harmless? And, since "non-harmless" changes are often necessary, how can one correctly compile and bootstrap them? 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 types and datatypes (not including functions, which are described by c and a) are laid out in memory.

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

b : Basis: the signature of the initial "pervasive" 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 pervasive basis.

2.1 Source characteristics

These characteristics are now used to describe the source code σ_1 for version 1 of the compiler:

$a_1 \in A_{\text{gen}}(\sigma_1)$ The compiler σ_1 contains a code generator for the a_1 architecture.

$C_{\text{gen}}(\sigma_1) = c_1$ The compiler σ_1 generates code that uses the c_1 calling conventions.

$D_{\text{gen}}(\sigma_1) = d_1$ The compiler σ_1 generates code that uses the d_1 datatype layout scheme.

$E_{\text{gen}}(\sigma_1) = e_1$ The compiler σ_1 uses (and writes to bin files) the e_1 static-environment representation.

$B_{\text{use}}(\sigma_1) = b_1$ The non-basis part of the compiler σ_1 is a program that makes use of functions in Basis b_1.

$B_{\text{imp}}(\sigma_1) = b_1$ The basis part of the compiler σ_1 implements the basis b_1.

$P_{\text{use}}(\sigma_1) = p_1$ The basis part of the compiler σ_1 is a program that makes use of the functions in Primitive basis p_1.

$P_{\text{gen}}(\sigma_1) = p_1$ The compiler σ_1 defines a primitive environment p_1 for its compiled code.

2.2 Binary file characteristics

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

$A_{\text{run}}(\beta_1) = a_1$ The program β_1 runs on architecture a_1.

$a_1 \in A_{\text{gen}}(\beta_1)$ The compiler β_1 generates code for the a_1 architecture.

$C_{\text{run}}(\beta_1) = c_1$ The program β_1 follows the c_1 calling conventions.

$C_{\text{gen}}(\beta_1) = c_1$ The compiler β_1 generates code that uses the c_1 calling conventions.

$D_{\text{run}}(\beta_1) = d_1$ The internal data structures of program β_1 obey the d_1 datatype layout scheme.

$D_{\text{env}}(\beta_1) = d_1$ The static environments in bin files β_1 use the d_1 datatype layout scheme.

$D_{\text{gen}}(\beta_1) = d_1$ The compiler β_1 generates code that uses the d_1 datatype layout scheme.

$E_{\text{env}}(\beta_1) = e_1$ The static environments bin files β_1 are in the e_1 environment representation.

$E_{\text{gen}}(\beta_1) = e_1$ The compiler β_1 uses and generates the e_1 environment representation.

$B_{\text{use}}(\beta_1) = b_1$ The non-basis part of the compiler β_1 is a program that makes use of functions in Basis (imports static pids) b_1.

$B_{\text{imp}}(\beta_1) = b_1$ The basis part of the compiler β_1 implements the basis (exports static pids) b_1.

$P_{\text{gen}}(\beta_1) = p_1$ 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 "basis") by the `makeml` script. Thus, the characteristics describing an executable file `sml-scb` (called μ here) 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 compilers generate code for only one machine, so $A_{\text{gen}}(\mu)$ is a single architecture rather than a set of architectures.

2.4 Runtime System characteristics

Finally, runtime systems are simple to characterize. Because they are not written in ML, there are no source $\rightarrow$ binary $\rightarrow$ source cycles. Thus, there is no need to distinguish between the source code and the executable of the runtime system.

$$A_{\text{run}}(\rho) = a, \qquad C_{\text{run}}(\rho) = c, \qquad D_{\text{run}}(\rho) = d$$

Runtime sysmtem ρ runs on architecture a, and interfaces to ML code with calling sequence c and datatype layout d.

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 5.

3.1 Compiling

To compile source code, one executes the "interactive batch compiler" `sml-scb` (which I will call μ) in source directory `src` (which corresponds to source σ), and performs `Batch.make()`. The result is a bin directory for the current architecture (if running on a Sparc, for example, `bin.sparc`) that I will call β.

This is abstracted as

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

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

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

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

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

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

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

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

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

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

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

$B_{\text{imp}}(\beta) = \mathcal{I}(B_{\text{imp}}(\sigma))$
where the $\mathcal{I}$ function computes intrinsic persistent identifiers (pids) from the static environment exported by a source program.[3] All one needs to know about $\mathcal{I}$ is that it is indeed a function, so $(x = y) \Rightarrow (\mathcal{I}(x) = \mathcal{I}(y))$.

[3] In version 0.97 of SML/NJ, the persistent identifers were generated from timestamps. Thus, `compile` was not really a function. This significantly complicated bootstrapping; see the footnote on page 5.

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

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

3.2 Bootstrapping

The usual way of building an executable `sml-scb` (denoted by μ) from runtime system source files `src/runtime` (ρ) and bin files `bin.sparc` (β) is with the shell commands

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

The `makeml` command builds an interactive system `sml`, and the `makesc` command loads the SC separate compilation system (in a special mode meant for compiling the compiler) into it, producing `sml-scb`.

I shall abstract this as

$$\texttt{makeml}(\rho, \beta) \;=\; \mu.$$

For this to work, the following equations must hold:

$A_{\text{run}}(\beta) = A_{\text{run}}(\rho) \in A_{\text{gen}}(\beta)$
so that the compiler, the runtime system, and top-level interactive commands are all running on the same computer.[4]

$C_{\text{run}}(\beta) = C_{\text{run}}(\rho)$
so the runtime system can call and be called by the ML program.

$C_{\text{run}}(\beta) = C_{\text{gen}}(\beta)$
so that top-level interactive commands can call and be called by the compiler and pervasive basis, and so that the SC subsystem can be compiled (from source files) into the ML system.

$D_{\text{run}}(\beta) = D_{\text{gen}}(\beta) = D_{\text{run}}(\rho)$
for the same reasons.

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

$E_{\text{env}}(\beta) = E_{\text{gen}}(\beta)$ for the same reason.

The remaining equations characterize the output μ:

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

$C_{\text{run}}(\beta) = C_{\text{run}}(\mu)$

$C_{\text{gen}}(\beta) = C_{\text{gen}}(\mu)$

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

[4] In compiling the compiler with SC, it may be that the only interesting "interactive" command that must be run in the same process is `Batch.make();` but there it is. It is also useful to have an interactive system to set various compiler flags before doing `Batch.make()`.

$D_{\text{gen}}(\beta) = D_{\text{gen}}(\mu)$

$E_{\text{gen}}(\beta) = E_{\text{gen}}(\mu)$

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

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

$P_{\text{gen}}(\beta) = P_{\text{gen}}(\mu)$

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

3.3 Retargeting

Because it is impossible to bootstrap using `compile` and `makeml` if the new compiler uses a new calling sequence or environment representation, two special procedures are provided. The first of these is called `retarget`: Execute some version μ_1 of `sml-scb`, and execute

$$\texttt{Retarget.retarget}(\beta, a);\ \ \texttt{exportML}(\mu_2)$$

where β is the name of a bin directory (`"bin.sparc"`, for example), a is the name of a machine architecture (`".mipseb"`, for example), and μ_2 is the name of a new sml-scb executable (`"sml-scb2"`, for example).

I will abstract this process as:

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

Retarget actually works by loading the compiler β into the executing μ_1 as if it were an ordinary user program. The compiler originally present in μ_1 will continue to exist (and be used for compiling top-level interactive commands), but the compiler β will be used for the implementation of `Batch.make()` (that is, `compile`).

Ordinary user programs do not provide their own implementation of the initial basis, so the basis portion of β (corresponding to source files from the `boot` directory, which implement $B_{\text{imp}}(\beta)$) will not be loaded. Thus, $B_{\text{imp}}(\beta)$ is irrelevant. However, the non-basis 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 doesn't need to be compatible with the instruction set or calling conventions that μ_1 itself is using. This freedom is crucial to the cross-compilation process (compilation for a different target architecture or calling convention).

The following restrictions apply:

$a \in A_{\text{gen}}(\beta)$

$A_{\text{run}}(\beta) = A_{\text{run}}(\mu_1)$

$C_{\text{run}}(\beta) = C_{\text{run}}(\mu_1)$

$D_{\text{run}}(\beta) = D_{\text{run}}(\mu_1)$

$D_{\text{env}}(\beta) = D_{\text{run}}(\mu_1)$

$E_{\text{env}}(\beta) = E_{\text{gen}}(\mu_1)$

$B_{\text{use}}(\beta) = B_{\text{imp}}(\mu_1)$

This last relationship is an *exact* signature match. In particular, it means that the intrinsic persistent identifiers generated from the compilation of the pervasive basis (files in the `src/boot` directory) in building the bin files within μ_1 must be identical to the corresponding identifiers in the pervasive basis portion of β. [5]

The following equations characterize the output μ_2:

$A_{\text{run}}(\mu_2) = A_{\text{run}}(\mu_1)$

$A_{\text{gen}}(\mu_2) = a$

$C_{\text{run}}(\mu_2) = C_{\text{run}}(\mu_1)$

$C_{\text{gen}}(\mu_2) = C_{\text{gen}}(\beta)$

$D_{\text{run}}(\mu_2) = D_{\text{run}}(\mu_1)$

$E_{\text{gen}}(\mu_2) = E_{\text{gen}}(\beta)$

$B_{\text{use}}(\mu_2) = B_{\text{use}}(\mu_1)$

$B_{\text{imp}}(\mu_2) = B_{\text{imp}}(\mu_1)$

$P_{\text{gen}}(\mu_2) = P_{\text{gen}}(\beta)$

A funny hybrid indeed.

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

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

where the source files of the `boot` subdirectories (source code for B_{imp}) of σ_1 and σ_2 are *identical*.

In versions of the SML/NJ compiler before 0.99, where the persistent identifiers (pids) were not intrinsic, even this would not guarantee equality of pids between $B(\beta)$ and $B(\beta_1)$. It was then required to create a new, empty bin directory; copy *just the pervasive bin files* from β_1 to the new bin directory β; and then proceed with a `compile` that would use these files as a starting point.

3.4 Elaboration

Finally, the `-elab` flag to `makeml` causes the static environment to be re-elaborated from source files, instead of read from the bin files. The process looks like:

`cd src; makeml -elab -bin bin.sparc; makesc`
which I will abstract as $\texttt{elab}(\rho, \beta, \sigma) = \mu$. Now, given the two steps

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

(ρ, β) must satisfy all the equations given for `makeml` 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.

The resulting executable μ is defined by the same equations as for $\texttt{makeml}(\rho, \beta)$.

4 Stable versions

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

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

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

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

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

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

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

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

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

Remark: If (ρ, σ, β) is a stable version, and

$$\texttt{compile}(\texttt{makeml}(\rho, \sigma, \beta), \sigma) = \beta'$$

then it is probably true that

$$\texttt{compile}(\texttt{makeml}(\rho, \sigma, \beta'), \sigma) = \beta'$$

(that is, (ρ, σ, β') is a fixed point), but this cannot be proved from the axioms in this paper.

5 New versions

Suppose one starts with a stable version $(\rho_1, \sigma_1, \beta_1)$ and creates a new source and runtime system ρ_2, 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 should be provable using only the axioms of section 3.

5.1 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

$$\begin{aligned} \texttt{makeml}(\rho_1, \beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \end{aligned}$$

The reader can verify that both of these steps succeed, and that $(\rho_1, \sigma_2, \beta_2)$ is stable.

5.2 New Pervasive Basis

Now suppose σ_2 differs from σ_1 in the pervasive basis (the files in the `boot` directory), and perhaps also in P_{use}, the set of primitives used in the pervasive basis.

$$\begin{aligned} \texttt{makeml}(\rho_1, \beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \end{aligned}$$

The reader can very that these steps succeed and result in $(\rho_1, \sigma_1, \beta_2)$ stable.

5.3 New environments

Suppose σ_2 uses a different environment representation (E_{gen}) from σ_1.

The "ordinary" procedure will not work:

$$\begin{aligned} \texttt{makeml}(\rho_1, \beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \\ \texttt{makeml}(\rho_1, \beta_2) &= \bot \end{aligned}$$

Now $E_{\text{gen}}(\beta_2) \neq E_{\text{env}}(\beta_2)$, so β_2 cannot be used in `makeml`. There are two choices:

$$\texttt{elab}(\rho_1, \beta_2, \sigma_2) = \mu_2$$

or,

$$\texttt{retarget}(\mu_1, \beta_2, A_{\text{run}}(\mu_1)) = \mu_2'$$

Now, with

$$\texttt{compile}(\mu_2, \sigma_2) = \beta_2'$$

then $(\rho_1, \sigma_2, \beta_2')$ is stable; or with

$$\texttt{compile}(\mu_2', \sigma_2) = \beta_2''$$

then $(\rho_1, \sigma_2, \beta_2'')$ is stable.

5.4 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 a new runtime system ρ_2 is also required, such that $D_{\text{run}}(\rho_2) = D_{\text{gen}}(\sigma_2)$.

Then the following steps will build a stable compiler:

$$\begin{aligned} \texttt{makeml}(\rho_1, \beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \\ \texttt{elab}(\rho_2, \beta_2, \sigma_2) &= \mu_2 \end{aligned}$$

It is left as an exercise for the reader to show that `elab` must be used, and that `retarget` will not suffice.

5.5 New calling conventions

Suppose σ_2 uses new calling conventions: $C_{\text{gen}}(\sigma_2) \neq C_{\text{gen}}(\sigma_1)$. Then a new runtime system ρ_2 is required, so that $C_{\text{gen}}(\sigma_2) = C_{\text{run}}(\rho_2)$.

The procedure is:

$$\begin{aligned} \texttt{makeml}(\rho_1, \beta_1) &= \mu_1 \\ \texttt{compile}(\mu_1, \sigma_2) &= \beta_2 \\ \texttt{retarget}(\mu_1, \beta_2, A_{\text{run}}(\mu_1)) &= \mu_2 \\ \texttt{compile}(\mu_1', \sigma_2) &= \beta_2' \end{aligned}$$

Now $(\rho_2, \sigma_2, \beta_2')$ is stable. The reader can verify that `retarget` is necessary, and that `elab` would not suffice.

5.6 New target architecture

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

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

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

5.7 Getting from here to there

Suppose there is a stable version $(\rho_1, \sigma_1, \beta_1)$, and a compiler $\mu_1 = \texttt{makeml}(\rho_1, \beta_1)$. The programmer makes a new source (and runtime system) σ_2, ρ_2 that differ in every characteristic from σ_1, ρ_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 $(\rho_2, \sigma_2, \beta_2)$ 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 $B_{\text{use}}(\sigma_2) \sqsubseteq B_{\text{imp}}(\sigma_2)$ or $P_{\text{use}}(\sigma_2) \sqsubseteq P_{\text{gen}}(\mu_1)$.

The procedures `makeml`, `retarget`, and `elab` are not useful, since they just take the binary files β_1 that we already have. $\texttt{elab}(\rho_2, \beta_1, \sigma_2)$ is illegal (as the reader may verify), and the author cannot even imagine why it might be useful.

Thus, to proceed towards a stable version involving σ_2 it is necessary to make an intermediate version σ_x as follows:

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

Then

$$\begin{aligned} \texttt{compile}(\mu_1, \sigma_x) &= \beta_x \\ \texttt{makeml}(\rho_1, \beta_x) &= \mu_x \end{aligned}$$

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

$$\begin{aligned} \texttt{compile}(\mu_x, \sigma_y) &= \beta_y \\ \texttt{makeml}(\rho_1, \beta_y) &= \mu_y \\ \texttt{compile}(\mu_y, \sigma_2) &= \beta_z \\ \texttt{retarget}(\mu_y, \beta_z) &= \mu_z \end{aligned}$$

$$\begin{aligned}\texttt{compile}(\mu_z, \sigma_2) &= \beta'_z \\ \texttt{elab}(\rho_2, \beta'_z, \sigma_2) &= \mu_2 \\ \texttt{compile}(\mu_2, \sigma_2) &= \beta_2\end{aligned}$$

Now $(\rho_2, \sigma_2, \beta_2)$ is stable. Verification of these steps, and of the stability of the final version, is left as an exercise.

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.

What this will not prove is that there is no shorter sequence of steps that will produce a β_2 that will be stable with μ_2, σ_2. 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.

6 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 faithfully implements the 100-page *Definition of Standard ML*[5] 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.

6.1 Structured I/O format

For example, John Reppy recently rewrote the "structure blaster" in the runtime system, that takes pointer data structures and writes them to (and reads them from) files. The structure blaster is used to write and read static environment representations from bin files β. Reppy's new structure blaster uses a different file format from the old one. The implementation of (either version of) the structure blaster, 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 $\texttt{makeml}(\rho, \beta) = \mu$ 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 $\texttt{retarget}(\mu_1, \beta, a) = \mu_2$ we have

$$\begin{aligned}F_{\text{env}}(\beta) &= F_{\text{gen}}(\mu_1) \\ F_{\text{gen}}(\mu_2) &= F_{\text{gen}}(\mu_1).\end{aligned}$$

And finally, for $\texttt{elab}(\rho, \beta, \sigma) = \mu$ 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 `makeml` 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.

6.2 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 in 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`, `makeml`, `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.

6.3 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. non-alphabetical) 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.

7 Related work

Lecarme *et al.* [4] present a good explanation of a theory of bootstrapping using T-diagrams, a notation invented by Bratman[2] and formalized by Earley and Sturgis[3]. This theory is simple, 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 5.1–5.6.

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 5.7.

Why do I abandon T-diagrams? The added problem in SML/NJ (and in similar systems that are interactive or have predigested type information with type-safe linking) is that there are extra constraints between the implementation language and the object language that 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 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.

8 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.

References

[1] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, page (to appear). ACM Press, June 1994.

[2] Harvey Bratman. An alternate form of the UNCOL diagram. *Communications of the ACM*, 4(3):142, March 1961. a one-page paper that is half diagrams!

[3] Jay Earley and Howard Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, October 1970.

[4] Olivier Lecarme, Mireille Pellissier, and Marie-Claude Thomas. Computer-aided production of language implementation systems: A review and classification. *Software—Practice and Experience*, 12(9):785–824, 1982.

[5] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1989.