Chapter 13

Bootstrapping a compiler

13.1 Introduction

When writing a compiler, one will usually prefer to write it in a high-level language. A possible choice is to use a language that is already available on the machine where the compiler should eventually run. It is, however, quite common to be in the following situation:

You have a completely new processor for which no compilers exist yet. Nevertheless, you want to have a compiler that not only targets this processor, but also runs on it. In other words, you want to write a compiler for a language A, targeting language B (the machine language) and written in language B.

The most obvious approach is to write the compiler in language B. But if B is machine language, it is a horrible job to write any non-trivial compiler in this language. Instead, it is customary to use a process called "bootstrapping", referring to the seemingly impossible task of pulling oneself up by the bootstraps.

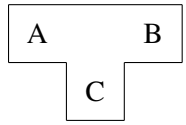
The idea of bootstrapping is simple: You write your compiler in language A (but still let it target B) and then let it compile itself. The result is a compiler from A to B written in B.

It may sound a bit paradoxical to let the compiler compile itself: In order to use the compiler to compile a program, we must already have compiled it, and to do this we must use the compiler. In a way, it is a bit like the chicken-and-egg paradox. We shall shortly see how this apparent paradox is resolved, but first we will introduce some useful notation.

13.2 Notation

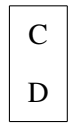
We will use a notation designed by H. Bratman [11]. The notation is hence called "Bratman diagrams" or, because of their shape, "T-diagrams".

In this notation, a compiler written in language C, compiling from the language A and targeting the language B is represented by the diagram



In order to use this compiler, it must "stand" on a solid foundation, *i.e.*, something capable of executing programs written in the language C. This "something" can be a machine that executes C as machine-code or an interpreter for C running on some other machine or interpreter. Any number of interpreters can be put on top of each other, but at the bottom of it all, we need a "real" machine.

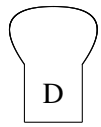
An interpreter written in the language D and interpreting the language C, is represented by the diagram



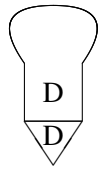
A machine that directly executes language D is written as

The pointed bottom indicates that a machine need not stand on anything; it is itself the foundation that other things must stand on.

When we want to represent an unspecified program (which can be a compiler, an interpreter or something else entirely) written in language D, we write it as

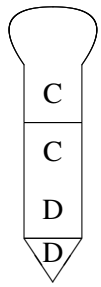


These figures can be combined to represent executions of programs. For example, running a program on a machine D is written as



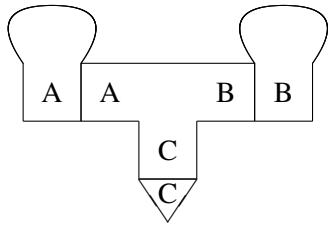
Note that the languages must match: The program must be written in the language that the machine executes.

We can insert an interpreter into this picture:



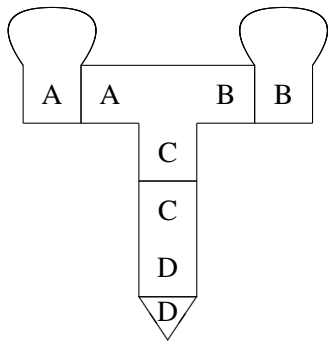
Note that, also here, the languages must match: The interpreter can only interpret programs written in the language that it interprets.

We can run a compiler and use this to compile a program:



The input to the compiler (*i.e.*, the source program) is shown at the left of the compiler, and the resulting output (*i.e.*, the target program) is shown on the right. Note that the languages match at every connection and that the source and target program are not "standing" on anything, as they are not executed in this diagram.

We can insert an interpreter in the above diagram:



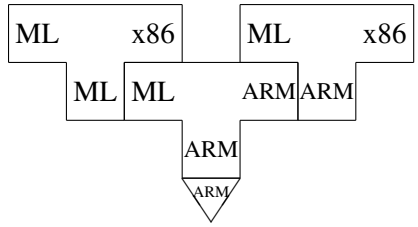
13.3 Compiling compilers

The basic idea in bootstrapping is to use compilers to compile themselves or other compilers. We do, however, need a solid foundation in form of a machine to run the compilers on.

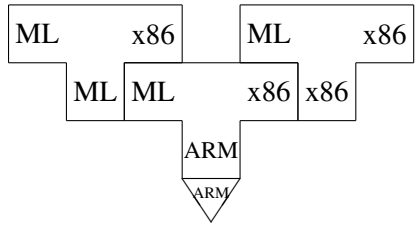
It often happens that a compiler does exist for the desired source language, it just does not run on the desired machine. Let us, for example, assume we want a compiler for ML to x86 machine code and want this to run on an x86. We have access to an ML-compiler that generates ARM machine code and runs on an ARM machine, which we also have access to. One way of obtaining the desired compiler would be to do *binary translation*, *i.e.*, to write a compiler from ARM machine code to x86 machine code. This will allow the translated compiler to run on an x86, but it will still generate ARM code. We can use the ARM-to-x86 compiler to translate this into x86 code afterwards, but this introduces several problems:

- Adding an extra pass makes the compilation process take longer.
- Some efficiency will be lost in the translation.
- We still need to make the ARM-to-x86 compiler run on the x86 machine.

A better solution is to write an ML-to-x86 compiler in ML. We can compile this using the ML compiler on the ARM:



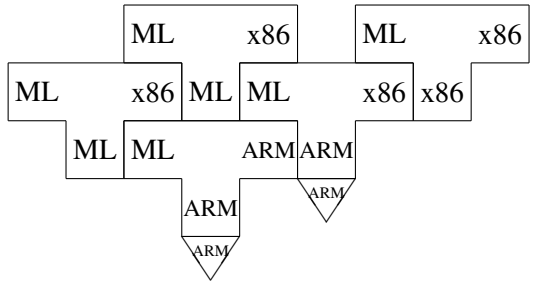
Now, we can run the ML-to-x86 compiler on the ARM and let it compile itself[1]:



We have now obtained the desired compiler. Note that the compiler can now be used to compile itself directly on the x86 platform. This can be useful if the compiler is later extended or, simply, as a partial test of correctness: If the compiler, when compiling itself, yields a different object code than the one obtained with the above process, it must contain an error. The converse is not true: Even if the same target is obtained, there may still be errors in the compiler.

[1] We regard a compiled version of a program as the same program as its source-code version.

It is possible to combine the two above diagrams to a single diagram that covers both executions:



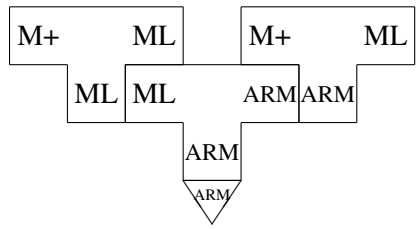
In this diagram, the ML-to-x86 compiler written in ARM has two roles: It is the output of the first compilation and the compiler that runs the second compilation. Such combinations can, however, be a bit confusing: The compiler that is the input to the second compilation step looks like it is also the output of the leftmost compiler. In this case, the confusion is avoided because the leftmost compiler is not running and because the languages do not match. Still, diagrams that combine several executions should be used with care.

13.3.1 Full bootstrap

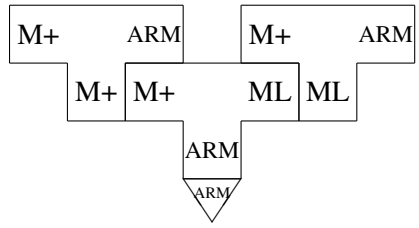
The above bootstrapping process relies on an existing compiler for the desired language, albeit running on a different machine. It is, hence, often called "half bootstrapping". When no existing compiler is available, *e.g.*, when a new language has been designed, we need to use a more complicated process called "full bootstrapping".

A common method is to write a QAD ("quick and dirty") compiler using an existing language. This compiler needs not generate code for the desired target machine (as long as the generated code can be made to run on some existing platform), nor does it have to generate good code. The important thing is that it allows programs in the new language to be executed. Additionally, the "real" compiler is written in the new language and will be bootstrapped using the QAD compiler.

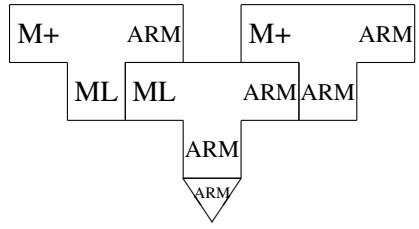
As an example, let us assume we design a new language "M+". We, initially, write a compiler from M+ to ML in ML. The first step is to compile this, so it can run on some machine:



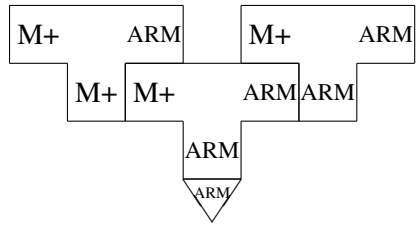
The QAD compiler can now be used to compile the "real" compiler:



The result is an ML program, which we need to compile:



The result of this is a compiler with the desired functionality, but it will probably run slowly. The reason is that it has been compiled by using the QAD compiler (in combination with the ML compiler). A better result can be obtained by letting the generated compiler compile itself:



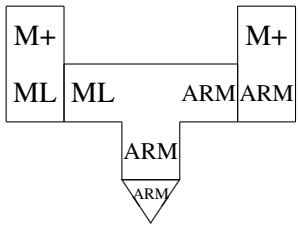
This yields a compiler with the same functionality as the above, *i.e.*, it will generate the same code, but, since the "real" compiler has been used to compile it, it will run faster.

The need for this extra step might be a bit clearer if we had let the "real" compiler generate x86 code instead, as it would then be obvious that the last step is

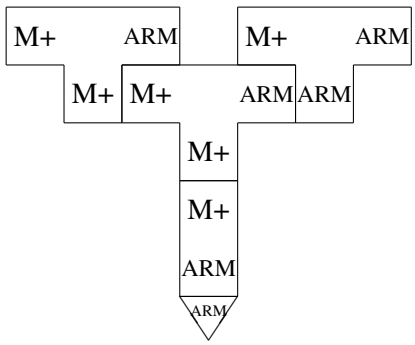
required to get the compiler to run on the same machine that it targets. We chose the target language to make a point: Bootstrapping might not be complete even if a compiler with the right functionality has been obtained.

Using an interpreter

Instead of writing a QAD compiler, we can write a QAD interpreter. In our example, we could write an M+ interpreter in ML. We would first need to compile this:



We can then use this to run the M+ compiler directly:



Since the "real" compiler has been used to do the compilation, nothing will be gained by using the generated compiler to compile itself, though this step can still be used as a test and for extensions.

Though using an interpreter requires fewer steps, this should not really be a consideration, as the computer(s) will do all the work in these steps. What is important is the amount of code that needs to be written by hand. For some languages, a QAD compiler will be easier to write than an interpreter, and for other languages an interpreter is easier. The relative ease/difficulty may also depend on the language used to implement the QAD interpreter/compiler.

Incremental bootstrapping

It is also possible to build the new language and its compiler incrementally. The first step is to write a compiler for a small subset of the language, using that same

subset to write it. This first compiler must be bootstrapped in one of the ways described earlier, but thereafter the following process is done repeatedly:

1) Extend the language subset slightly.

2) Extend the compiler so it compiles the extended subset, but without using the new features.

3) Use the previous compiler to compile the new.

In each step, the features introduced in the previous step can be used in the compiler. Even when the full language is compiled, the process can be continued to improve the quality of the compiler.

Suggested exercises: 13.1.

13.4 Further reading

Bratman's original article, [11], only describes the T-shaped diagrams. The notation for interpreters, machines and unspecified programs was added later in [15].

An early bootstrapped compiler was LISP 1.5 [30].

The first Pascal compiler [45] was made using incremental bootstrapping.

Though we in section 13.3 dismissed binary translation as unsuitable for porting a compiler to a new machine, it is occasionally used. The advantage of this approach is that a single binary translator can port any number of programs, not just compilers. It was used by Digital Equipment Corporation in their FX!32 software [18] to enable programs compiled for Windows on an x86-platform to run on their Alpha RISC processor.

Exercises

Exercise 13.1

You have a machine that can execute *Alpha* machine code and the following programs:

1: A compiler from C to *Alpha* machine code written in *Alpha* machine code.

2: An interpreter for ML written in C.

3: A compiler from ML to C written in ML.

Now do the following:

a) Describe the above programs and machine as diagrams.

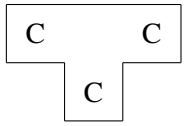
b) Show how a compiler from ML to C written in *Alpha* machine code can be generated from the above components. The generated program must be stand-alone, *i.e.*, it may not consist of an interpreter and an interpreted program.

c) Show how the compiler generated in question b can be used in a process that compiles ML programs to *Alpha* machine code.

Exercise 13.2

A source-code optimiser is a program that can optimise programs at source-code level, *i.e.*, a program O that reads a program P and outputs another program P', which is equivalent to P, but may be faster.

A source-code optimiser is like a compiler, except the source and target languages are the same. Hence, we can describe a source-code optimizer for C written in C with the diagram



Assume that you additionally have the following components:

- A compiler, written in ARM code, from C to ARM code.
- A machine that can execute ARM code.
- Some unspecified program P written in C.

Now do the following:

a) Describe the above components as diagrams.

b) Show, using Bratman diagrams, the steps required to optimise P to P' and then execute P'.