# Compiling Scheme to a Micro$^2$-computer

## 1 Micro-squared-computers

In the Theory of Computation, it's useful to study models of extremely simplified general-purpose computers. These machine models fall off the minimum end of the design charts even for a RISC (Reduced Instruction Set Computer) *micro*computer chip, so we'll call them *micro*$^2$-computers to emphasize this point.

Turing machines are the traditional micro$^2$-computer used in the development of Computability Theory. In these notes, we'll outline how programs written in Scheme can be compiled to run on a micro$^2$-computer that is arguably simpler than a classical Turing machines. The target machine for our Scheme compiler will be a *Counter Machine* with two counters.

Of course even though in theory Scheme can be compiled into micro$^2$-computer code, we are not suggesting that this would be a sensible thing to do. The resulting compiled code is spectacularly inefficient, and there would be no point in trying to run it. But construction of the compiler fleshes out the assertion that any procedure programmable in Scheme can be simulated by a micro$^2$-computer program. Consequently, to prove that some decision procedure can be carried out by a micro$^2$-computer, it's sufficient to demonstrate it can be done in Scheme. In fact, we have methods for designing an interpreter for any other programming language in Scheme, so if we see how to write a procedure in *any* programming language, we can construct a micro$^2$-computer to carry out the procedure.

Compiling Scheme to micro$^2$-computers also explains why analyzing the computational behavior of micro$^2$-computer programs is going to be hard in general. Even though they look very limited, analyzing micro$^2$-computer programs is going to be as challenging as analyzing arbitrary procedures in a full-fledged programming language. This is a daunting task—think of trying to understand an undocumented procedure definition consisting of a few thousand lines of "spaghetti code". Determining whether programs even have some simple computational properties is in fact *undecidable*, as we have discussed in other Notes. So significantly paring down the features in a programming language may not guarantee that the pared-down programs will be easy to analyze.

But the main reason to compile to micro$^2$-computers is that it simplifies explaining how various problems from outside Computer Science can be understood as general computation problems in disguise. Recognizing the computational component in problems from algebra, number theory, formal logic, formal grammars, and combinatrial optimization, for example, will allow us to conclude that undecidable properties carry over to these disciplines.

For example, the negative solution to Hilbert's Tenth Problem shows that the set of multivariable integer-coefficient polynomials with an integer root-vector is not a decidable set. The proof of

| instruction | informal meaning |
|---|---|
| `inc`$i$ | increment Counter $i$ |
| `dec`$i$ | decrement Counter $i$, except leave it alone if it is 0. |
| `ifr`$i$ $n_1$ $n_2$ | if Counter $i$ is zero, go to line $n_1$, otherwise go to line $n_2$. |

Figure 1: Counter Machine Instructions

```
0:  ifr1 4 1   ;going to line 4 means "halt"
1:  dec1
2:  inc2
3:  ifr1 0 0   ;goto line 0
```

Figure 2: 2-Counter Machine $C_{\text{add}}$

this result involves learning to "program" with such polynomials. But polynomials are a pretty cumbersome language to program in, so the burden of showing how polynomials can describe arbitrary computations is a lot lighter when all we have to show is how polynomials can simulate some micro$^2$-computer.

## 2 Counter Machines

An $n$-Counter machine has $n$ registers, each register capable of holding a nonnegative integer of arbitrary size. The only operations of the machine are to increment (add one) to any register, decrement any register (leaving it unchanged if the register contains zero), and branch on whether a register contains zero, as indicated in Figure 1. The registers are called "counters" since in a single step they can only be incremented and decremented.

A Counter Machine, $C$, is a finite sequence of such instructions, numbered consecutively starting at zero. For example, the machine, $C_{\text{add}}$, in Figure 2 is a 2-Counter Machine that adds the contents of its first counter to its second counter and then halts (by transferring to an instruction number larger than any in the program).

The "configuration" of a 2-CM is defined to be a triple $(k, l, m)$ of nonnegative integers. The intended interpretation of the triple is that $k$ is the number of the next instruction to be executed, and $l$ and $m$ are the contents of Counters 1 and 2. Formally, we associate with 2-CM, $C$, a partial function $\text{step}_C$ from configurations to configurations, where $\text{step}_C(k, l, m)$ is the configuration, if any, after execution of $C$'s $k$th instruction when Counter 1 contains $l$, and Counter 2 contains $m$. For example, for the machine $C_{\text{add}}$ of Figure 2:

$$\begin{aligned}
\text{step}_{C_{\text{add}}}(0, 2, 0) &= (1, 2, 0), \\
\text{step}_{C_{\text{add}}}(1, 2, 0) &= (2, 1, 0), \\
\text{step}_{C_{\text{add}}}(2, 1, 0) &= (3, 1, 1), \\
\text{step}_{C_{\text{add}}}(3, 1, 1) &= (0, 1, 1).
\end{aligned}$$

**Problem 1.** Give a precise definition of the function $\text{step}_C$ for an arbitrary 2-CM, $C$.

## 3 Compiling to Counter Machines

Given the limited structure of counter machines, it is not completely apparent what it means to compile a Scheme expression into one of them. For example, Scheme values such as procedures, symbols, lists, or even *negative* integers are not available in counter machines, so we cannot expect a target counter machine program to produce the same values as an arbitrary source Scheme expression.

One approach would be to code Scheme values into nonnegative integers. We would then require that for any Scheme "source" expression, $E$, the compiled counter machine $C_E$, started in some standard way—say with an all zero configuration—halts with the the nonnegative integer representation of the value of $E$ in some designated counter. However, for theoretical purposes it is not necessary to develop such a representation of Scheme values. It is enough to capture whether the Scheme evaluation—started from the standard user's initial environment—"halts" by returning some value; we need not specify what the value, if any, actually is.

Our main result is:

**Theorem 3.1.** *(Scheme→2-CM) There is a procedure for translating any Scheme expression, $E$, into a 2-counter machine, $C_E$, such that*

$$E \text{ returns a value iff } C_E \text{ halts.}$$

## 4 From Scheme to Register Machines

Going from Scheme expression, $E$, to the 2-CM, $C_E$, is a long trip, and we need some "rest stops" along the way. The beginning leg of the trip is the longest, ending at the first rest stop: *List Register Machines* (LM's).

LM's are the machines which Abelson/Sussman use as a target for the Scheme compiler described in their book. These are machines with a fixed number of registers capable of holding primitive values like symbols and numbers, or `cons`-cells whose `car`'s and `cdr`'s may be primitive values or other `cons`-cells.

In their initial development of LM's, Abelson/Sussman include some moderately high-level operations on environments in the instruction set, e.g., `lookup-in-environment`. Subsequently, they explain how to implement all such operations using basic operations on primitive data types like integers, booleans, and symbols, along with only the list operations:

`cons`, `car`, `cdr`, `set-car!`, `set-cdr!`, `null?`.

The second rest stop is also described by Abelson/Sussman. They show how to represent the "heap" of `cons`-cells and their contents[1] using Random Access Machines (RAM's). These are machines with an array of registers numbered consecutively starting at zero. Each register may hold

[1] They also explain how garbage-collection works for the heap, but for our theoretical purposes, garbage-collection is not necessary.

an arbitrary nonnegative integer. The RAM instruction set involves familiar arithmetic operations on register contents. Registers may be referenced *directly*, as in the assignment statement:

```
R4 := R4 + R5
```

Executing this instruction changes the contents of Register 4 to be the sum of its current contents and the contents of Register 5.

Registers may also be referenced *indirectly*, as in

```
ref(R4) := R3 * ref(R6)
```

Here Registers 4 and 6 have indirect references. Specifically, let $m_4$ be the contents of Register 4, and $m_6$ the contents of Register 6. Executing this instruction sets the contents of Register $m_4$ to the product of the contents of Register 3 and the contents of Register $m_6$.

Test instructions like

```
if 4 < ref(R1) goto line 7 else goto line 9
```

may also refer to registers directly or indirectly. A RAM is defined to be a finite sequence of such instructions.

Formally, we can define a configuration of a RAM to be a sequence

$$\langle c, n_0, n_1, n_2, \ldots, n_k \rangle$$

of $k+2$ nonnegative integers, where $c$ specifies the number of the next instruction to be executed, $n_0, n_1, \ldots, n_k$ are the contents of Registers 0, 1, ..., $k$, and all registers numbered greater than $k$ contain zero.

For definiteness, we may assume that the integer operations of RAM's are exactly `+`, `*`, `-` (restricted to return zero if a result would be negative), the test operations are exactly `<` and `=`, and all instructions are of one of the forms illustrated above.

**Problem 2.** Let $R$ be a RAM. Give a precise definition of the function $\text{step}_R$ mapping a configuration of $R$ to the "next" configuration, if any, that follows after executing the designated next instruction of $R$.

So with Abelson/Sussman as our guides on these first legs of the trip, we arrive at

**Theorem 4.1.** *(Scheme→RAM). There is a procedure for translating any Scheme expression, $E$, into a RAM, $R_E$, such that*

*$E$ returns a value iff $R_E$ halts.*

This translation is normally referred to as compilation, and the Abelson/Sussman compiled version, $R_E$, of $E$ will exhibit all the computational behavior of $E$. But as explained above, for our purposes Theorem Scheme 4.1 is all we need to know about "real" compilation.

# 5 Register Machines

The next rest stop will be "ordinary" Register Machines (RegM's). These are the same as RAM's except that indirect register references are not allowed. This means that only a *fixed* number of registers can be accessed by a given RegM.

A RAM, $R$, can be simulated with a RegM, $M_R$, by coding the entire current configuration of $R$ into *one* number that $M_R$ stores in some register. Let *pair* be some function which codes any pair of nonnegative integers into a single integer, e.g,

$$\text{pair}(m, n) = 2^m 3^n.$$

Now code a configuration $\langle c, n_0, n_1, \ldots, n_k\rangle$ of $R$ into the single number, $\text{rep}(\langle c, n_0, n_1, \ldots, n_k\rangle)$, defined as:

$$\text{rep}(\langle c, n_0, n_1, \ldots, n_k\rangle) ::= \text{pair}(k, \text{pair}(c, \text{pair}(n_0, \ldots, \text{pair}(n_{k-1}, n_k)\ldots))).$$

It is not hard to program a RegM to update

$$\text{rep}(\langle c, n_0, n_1, \ldots, n_k\rangle)$$

into

$$\text{rep}(\text{step}_R(\langle c, n_0, n_1, \ldots, n_k\rangle))$$

as suggested by 3. In fact, besides the register used to store the number representing the $R$'s configuration, $M_R$ only needs some small fixed number of registers—not more than four—for various temporary results. In other words, we can be sure that $M_R$ is a 4-Register Machine, no matter how many registers $R$ may access.

**Problem 3.** **(a)** Exhibit a RegM, $M_{\text{pair}}$, which, started in configuration, $\langle 0, k, l\rangle$, halts in configuration $\langle n, m\rangle$ where $m = \text{pair}(k, l)$ and $n = \text{length}(M_{\text{pair}})$.

**(b)** Exhibit a RegM, $M_{\text{fetch}}$, which, started with $\text{rep}(\langle c, n_0, n_1, \ldots, n_k\rangle)$ in Register 1 and an integer $m_2$ in Register 2, halts with Registers 1 and 2 unchanged, and the integer $n_{m_2}$ in Register 3 if $m_2 \leq k$. $M_{\text{fetch}}$ should halt with zero in Register 3 if $m_2 > k$.

So we have

**Theorem 5.1.** *(RAM→RegM). There is a procedure for translating any RAM, $R$, into a 4-Register Machine, $M_R$, such that*

*$R$ halts iff $M_R$ halts.*

# 6 Register Machines to Counter Machines

It is easy to see how to write RegM routines which add, subtract, multiply, compare, or copy registers using only increment, decrement, and branch on zero instructions, which leads to our penultimate rest stop:

**Theorem 6.1.** *(RegM→CM). There is a procedure for translating any Register machine, $M$, into an Counter machine, $C_M$, such that*

*$M$ halts iff $C_M$ halts.*

**Problem 4.** **(a)** Exhibit an counter machine which, started in configuration $\langle 0, m, n, 0, \ldots, 0\rangle$ halts in a configuration of the form $\langle l, m, n, m+n, 0, \ldots, 0\rangle$.

**(b)** Same as the previous part with $m + n$ replaced by $m \cdot n$.

Finally, a $k$-counter machine, $K$, can be simulated by a 2-counter machine, $C_K$, that keeps in its first counter the code number representing the contents $n_1, n_2, \ldots, n_k$ of $K$'s counters. The code number will be

$$2^{n_1}3^{n_2}\cdots {p_k}^{n_k}$$

where $p_k$ is the $k$th prime number. To simulate an increment instruction on $K$'s Counter 2, for example, $C_K$ need only multiply the code number by the 3. This it can do by repeatedly decrementing its first counter once, while incrementing its second counter three times, until the first counter is empty. Then it copies the second counter back into the first by repeatedly decrementing the second and incrementing the first until the second is empty.

To simulate $K$'s testing Counter 3 for zero, $C_K$ can divide the code number by $p_3$, namely 5, and branch on the remainder. This it can do by repeatedly decrementing its first counter five times while incrementing its second counter once until the first counter is empty. At this point the quotient of the code number divided by 5 is in Counter 2, and number of decrements left unperformed among the final group of five is the remainder. Before branching on the remainder, $C_K$ can restore the code number into counter 1 by reversing the divide-by-5 process it just performed.

This sketch should be enough to understand how to complete the final leg of our trip:

**Theorem 6.2.** *($n$-CM→2-CM) There is a procedure for translating any $n$-Counter machine, $K$, into an 2-Counter machine, $C_K$, such that*

*$K$ halts iff $C_K$ halts.*

Combining all the legs of our trip completes the journey from Scheme to 2-CM's and the proof of Theorem 3.1.

**Problem 5.** Let $E$ be a Scheme expression whose value is a procedure of one argument. Explain how to construct a 3-CM, $C_E$, such that `(`$E$ `n)` returns a value iff $C_E$ halts when started in configuration $\langle 0, n, 0, 0\rangle$.

**Problem 6.** Prove that the halting problem for counter machines with only increment and branch-on-zero instructions—no decrement instructions—is decidable.