Programming Languages and Computation

Problem Sheet 8: Verifying and Compiling While Programs

We first reason about given While programs, and prove some simple properties.
We then compile While programs down to the Abstract Machine, and beyond.

Compiling While Programs

** 1. Consider the While programs from the Week 5 worksheet (also used in Week 6). Manually compile them to the Abstract Machine language.

You can test your attempts with the `bam` program provided in the labcode archive. Don't forget to initialize the variables meant to serve as inputs before running tests.

At the moment, `bam` reads its input (an Abstract Machine program) from `stdin` and prints any output (final state, or error messages) to `stdout`. In order to invoke it, run:

```
$ cat <filepath> | cabal run bam
```

or

```
$ echo "program" | cabal run bam
```

You can combine `echo` and `cat` to prepend any state initialisation required to pass arguments. For example, the following command will run the factorial program in a state with variable `n` initialised to 12.

```
$ echo "PUSH 12; STORE n;" | cat - bam/test/factn.am | cabal run bam
```

Making Haskell Compile While Programs

In this section, we compile While programs down to the abstract machine's language. (Or rather, we make Haskell do it for us.) Our compiler will translate While ASTs to AM ASTs in a syntax-directed way.

We do so in file while/src/WhileC.hs.

We first write a function to compile arithmetic expressions (type `AExpr`) down to Abstract Machine code (type `Code`).

An Abstract Machine program is just a list of instructions (although some of them currently have more structure), so we can construct them mostly using operations over type `List`. However, those are most efficient when constructing lists from right to left. (Another story of left recursion vs right recursion.)

To allow us to keep thinking about the compilation problem from left to right without losing the (very minor) performance gains of constructing the list from right to left, we write our translator using an accumulator. (In essence, we construct the AM program on our way back up from the recursive calls instead of constructing its beginning, then recursing down to construct the end.)

As an example, you may have already seen how the following naive implementation of list reversal (which has quadratic complexity)

```
rev :: [a] → [a]
rev [] = []
rev (x : xs) = (rev xs) ++ x
```

can be implemented much more efficiently (with linear complexity) as

```
rev :: [a] → [a]
rev = aux []
 where
  aux acc [] = acc
  aux acc (x : xs) = aux (x : acc) xs
```

** 2. The following makes you write the code in its most efficient right-to-left version directly. Feel free to first implement the compiler directly as specified in the reference material, then consider how to make it more efficient by avoiding repeated concatenations with increasingly long prefixes.

(a) Write function `arithToInstrs :: AExpr → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material). When its second argument is some list `c` of AM instructions, then its result is simply prepended to `c`.

(b) Write function `boolToInstrs :: BExpr → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material).

(c) Write function `stmtToInstrs :: Stmt → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material).

Compiling in Restricted Settings

Now that you understand compilation enough to implement it effectively, we can consider what happens if we try to compile to a slightly more restricted language.

We consider a knock-off Abstract Machine (the Knock-Off Machine, below) that can only hold two elements on its stack. We give (part of) its structural operational semantics as a transition relation $\Rightarrow_{KO}$ (Figure 1).

$$\begin{array}{rl}
\langle \texttt{PUSH}\ v; c, s, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, v : s, \sigma\rangle & \text{if } |s| \leq 1 \\
\langle \texttt{ADD}; c, z_2 : z_1, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, z_1 + z_2, \sigma\rangle & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \texttt{MUL}; c, z_2 : z_1, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, z_1 * z_2, \sigma\rangle & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \texttt{SUB}; c, z_2 : z_1, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, z_1 - z_2, \sigma\rangle & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \texttt{LOAD x}; c, s, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, \sigma(\texttt{x}) : s, \sigma\rangle & \text{if } |s| \leq 1 \\
\langle \texttt{STORE x}; c, z : s, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, s, [\texttt{x} \mapsto z]\sigma\rangle & \text{if } z \in \mathbb{Z}
\end{array}$$

Figure 1: Semantics for the Knock-Off Machine's arithmetic instructions

*** 3. (From last year's exam.) Define and argue correct a translation function $\mathscr{CA}_{\mathsf{KO}}[\![\cdot]\!]$ from While's *arithmetic expressions only* to the Abstract Machine's bytecode language.
When the source expression a has semantics $[\![\mathtt{a}]\!]_{\mathscr{A}}\,\sigma$ in a given state σ, executing the translated program $\mathscr{CA}_{\mathsf{KO}}[\![a]\!]$ with the empty evaluation stack in state σ should yield a configuration with an empty program, a stack containing exactly one element whose value is $[\![\mathtt{a}]\!]_{\mathscr{A}}\,\sigma$, and a state that must be larger than σ in the sense that it is defined and equal to σ wherever σ is defined, and may also give values to variables not defined in σ. You may choose to extend the domain of the state to inputs that are not valid While variables.
Your translation function may rely on an environment.

**** 4. (Optional.) The code repository also contains code for `blam`, an interpreter for the Bristol Less Abstract Machine (BLAM). The BLAM's language is similar to the AM's language, but eschews structured control-flow. Unlike the jump-based abstract machine seen in W8V6, the BLAM does not use labels, instead using `GOTO` and `GOTOF` to (conditionally) jump to the specified line number. Line numbers are 0-indexed (this is just to be slightly annoying).

Implement a compiler from While to BLAM. You can do so either by adapting the existing `whilec` compiler (from Q2), or by implementing a compiler from BAM to BLAM and composing compilation from While to BAM with compilation from BAM to BLAM.

You may want to start by defining semantics for BLAM (you can take a look at the source code to figure it out; we implement configurations of the form $\langle c, pc, s, \sigma\rangle$ by keeping track of the program's code as two lists that are split just before the next instruction to execute. This makes jumps a bit costly, but most linear code a lot cheaper.

Reasoning about While Programs

Consider the While program example from Week 1.

```
x := x * x - 1
while !(x = 0) {
  x := x - 2
  y := 2 * y - 1
}
```

* 5.

(a) Specialize the semantics of While to consider only configurations reachable from initial configurations whose statement component is the While program example.

(b) Compare the resulting set of transitions to the transition system described in the original example. (In such a situation, we say that the transition systems *simulate* each other.)

Practicing Backwards Reasoning. Let us now practice reasoning about programs, as illustrated in W8V1. We do so on the implementation S_{div} of Euclidean division shown in Figure 2.

```
q := 0
while (b <= a) {
  q := q + 1
  a := a - b
}
r := a
```

Figure 2: The Euclidean division program S_{div}.

Given an initial state σ such that $0 \leq \sigma(\mathtt{a})$ and $0 <\leq \sigma(\mathtt{b})$, there always exists a state σ' such that $\langle S_{\mathsf{div}}, \sigma\rangle \Rightarrow \left\langle \mathtt{skip}, \sigma' \right\rangle$, $\sigma(\mathtt{a}) = \sigma'(\mathtt{q}) \cdot \sigma(\mathtt{b}) + \sigma'(\mathtt{r})$, and $0 \leq \sigma'(\mathtt{r}) < \sigma(\mathtt{b})$.

*** 6. (Optional.) Using the backwards reasoning technique illustrated in the lecture, prove that the correctness property stated above holds.

Ask the lecturers for a loop invariant if you're having trouble finding it, but recall that a loop invariant must:

- Hold at the start of the first loop iteration;
- Be preserved by the loop; and
- Be sufficiently strong after the last loop iteration to ensure correctness of the whole program.

It is often useful to consider the property implied on the loop exit configuration, consider how it is transformed by one loop iteration (propagating backwards as seen in the lecture), and attempting to generalize from there.