G52AFP Coursework 2
Monadic Compiler

Graham Hutton
University of Nottingham

Abstract

The aim of this coursework is to write a monadic compiler that translates programs in a small imperative language into code for a simple stack machine, and to write a simulator for this machine.

Instructions

- This coursework counts for 15% of the assessment for the module, and may either be solved on your own, or jointly with **ONE** other student taking the module. Larger teams are not permitted. Students must not make their code publically available online.
- Solutions must be in the form of a literate Haskell script (.lhs document), and include a brief explanation for each definition. Bonus marks are available for particularly clear, simple, or elegant solutions.
- For identification purposes, your script must begin as follows:

  ```
  G52AFP Coursework 2 - Monadic Compiler

  Your full name(s)
  Your full email address(es)
  ```

  In the case of jointly produced solutions, only one copy should be submitted, containing the names of both students.
- The deadline for submission is 3pm on **WEDNESDAY 6th MAY**. Submission is electronic, via the unix-based **cw** system:

  ```
  https://support.cs.nott.ac.uk/coursework/cwstud/
  ```

Imperative language

Consider an imperative language in which a program is either an assignment, a conditional, a while loop, or a sequence of programs:

$$
\begin{aligned}
\textbf{data}\ \mathit{Prog} = {} & \mathit{Assign}\ \mathit{Name}\ \mathit{Expr} \\
\mid {} & \mathit{If}\ \mathit{Expr}\ \mathit{Prog}\ \mathit{Prog} \\
\mid {} & \mathit{While}\ \mathit{Expr}\ \mathit{Prog} \\
\mid {} & \mathit{Seqn}\ [\mathit{Prog}]
\end{aligned}
$$

In turn, an expression is either an integer value, a variable name, or the application of an operator to two argument expressions:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr} && = \mathit{Val}\ \mathit{Int} \mid \mathit{Var}\ \mathit{Name} \mid \mathit{App}\ \mathit{Op}\ \mathit{Expr}\ \mathit{Expr} \\
&\textbf{type}\ \mathit{Name} && = \mathit{Char} \\
&\textbf{data}\ \mathit{Op} && = \mathit{Add} \mid \mathit{Sub} \mid \mathit{Mul} \mid \mathit{Div}
\end{aligned}
$$

Note that the logical value False is represented by the integer zero, and True is represented by any other integer. For example, a program that computes the factorial of a non-negative integer n can be defined as follows:

```
fac   :: Int -> Prog
fac n = Seqn [Assign 'A' (Val 1),
              Assign 'B' (Val n),
              While (Var 'B') (Seqn
                [Assign 'A' (App Mul (Var 'A') (Var 'B')),
                 Assign 'B' (App Sub (Var 'B') (Val 1))])]
```

Virtual machine

Now consider a virtual machine that operates using a stack of integers, and a memory that maps variable names to their current integer values:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Stack} && = [\mathit{Int}] \\
&\textbf{type}\ \mathit{Mem} && = [(\mathit{Name}, \mathit{Int})]
\end{aligned}
$$

Code for the machine comprises a list of instructions, each of which either pushes an integer onto the stack, pushes the value of a variable, pops the top of the stack into a variable, performs an operation on the stack, jumps to a label, pops the stack and jumps if this value is zero, or is simply a label:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Code} && = [\mathit{Inst}] \\
&\textbf{data}\ \mathit{Inst} && = \mathit{PUSH}\ \mathit{Int} \\
& && \mid \mathit{PUSHV}\ \mathit{Name} \\
& && \mid \mathit{POP}\ \ \mathit{Name} \\
& && \mid \mathit{DO}\ \ \ \mathit{Op} \\
& && \mid \mathit{JUMP}\ \mathit{Label} \\
& && \mid \mathit{JUMPZ}\ \mathit{Label} \\
& && \mid \mathit{LABEL}\ \mathit{Label} \\
&\textbf{type}\ \mathit{Label} && = \mathit{Int}
\end{aligned}
$$

Exercise

Define a function $comp :: Prog \rightarrow Code$ that translates a program into machine code, using a state monad to handle the generation of fresh labels. For example, the result of $comp\ (fac\ 10)$ should be as follows:

```
[PUSH 1, POP 'A',
 PUSH 10, POP 'B',
   LABEL 0,
   PUSHV 'B', JUMPZ 1,
   PUSHV 'A', PUSHV 'B', DO Mul, POP 'A',
   PUSHV 'B', PUSH 1, DO Sub, POP 'B',
   JUMP 0,
 LABEL 1]
```

Exercise

Define a function $exec :: Code \rightarrow Mem$ that executes code produced by your compiler, returning the final contents of the memory. For example, the result of $exec\ (comp\ (fac\ 10))$ should be as follows:

```
[('A', 3628800), ('B', 0)]
```

Bonus

Define a function that attempts to optimise programs to execute in fewer steps. *Hint:* think about program-level vs code-level transformation, and/or extending the machine with extra instructions.

Revise your compiler to use the *writer monad* to handle the production of the resulting code. *Hint*: you will need to use the writer monad transformer, supplied with the state monad as an argument. (More difficult)