ECM2418: Programming Languages and Representations

Continuous Assessment (v21.1)

Diego Marmsoler

Due date: Thursday 9th December 2021

This continuous assessment (CA) is worth 40% of your final mark and intended to last for 40 hours. You can get up to 100 marks in total split across four exercises:

- A low-level execution machine (10h, 25 marks)
- A high-level programming language (10h, 25 marks)
- A basic compiler (10h, 25 marks)
- Verifying compiler correctness (10h, 25 marks)

The CA comes with a template Haskell project which you can download from the assessment section on the ELE page. The file `README.md` contains detailed instructions on how to build the project and run the tests using the tool stack [1]. Before submitting your coursework make sure the project compiles and the tests provided in the template succeed.

[1] `https://docs.haskellstack.org/en/stable/README/`

Contents

1 A low-level Execution Machine

For the following exercise you should implement a machine to execute low-level programming code.

1.1 Machine Specification

The machine operates on integer values and maintains a configuration which consists of:

- A program counter which points to the next statement to be executed.
- A stack which keeps a temporary list of values.
- A state which keeps the value of variables.

The machine is able to execute the following instructions to modify a given configuration:

LOADI x	Loads a value x onto the stack and increments the program counter by one.
LOAD v	Loads the value of a variable v onto the stack and increment the program counter by one.
ADD	Adds the two topmost values of the stack and increment the program counter by one.
STORE v	Stores the top of the stack to variable v and increments the program counter by one.
JMP i	Increments the program counter by i.
JMPLESS i	Compares the two topmost values x (the top element) and y (the element right after x) of the stack and in the case $y < x$, increments the program counter by i.
JMPGE i	Similar to JMPLESS but compares $y \geq x$ instead.

1.2 Tasks

Implement the machine by filling out the template `src/Machine.hs` as follows:

1. Define a type Vname to model variable names as strings.
2. Define a type Val to model variable values.
3. Define a type State for states which maps variable names to values.
4. Create a data type Instr which models all instructions supported by the machine.

5. Define a type Stack to model the stack.
6. Define a type Config to model a configuration.
7. Define a function iexec :: Instr → Config → Config to execute a single instruction.
8. Define a function exec :: [Instr] → Config → Config to execute a list of instructions.

1.3 Examples

The following examples demonstrate the execution of different instructions.

```
>iexec (LOADI 5) (0, empty, [])
>(1,fromList [],[5])
>
>iexec (LOAD "v1") (0, fromList [("v1",5)], [])
>(1,fromList [("v1",5)],[5])
>
>iexec ADD (0, empty, [5,6])
>(1,fromList [],[11])
>
>iexec (STORE "x") (0, empty, [5])
>(1,fromList [("x",5)],[])
>
>iexec (JMP 5) (0, empty, [])
>(6,fromList [],[])
>
>iexec (JMPLESS 5) (0, empty, [5,6])
>(1,fromList [],[])
>
>iexec (JMPGE 5) (0, empty, [5,6])
>(6,fromList [],[])
>
>exec [LOADI 1, LOADI 2, ADD] (0,empty,[])
>(3,fromList [],[3])
>
>exec [LOADI 1, STORE "v1", LOADI 2, STORE "v2"] (0,empty,[])
>(4,fromList [("v1",1),("v2",2)],[])
```

1.4 Notes

- Use the data type Map from the package `Data.Map` to model a state.
- fromList used in the examples above is a function which can be used to create a map from a list of values.

2 A high-level Programming Language

For the following exercise you need to implement an interpreter for a simple, imperative programming language.

2.1 Language Specification

The language consists of *commands* involving simple *arithmetic* and *boolean expressions*.

2.1.1 Arithmetic expressions

It supports the following arithmetic expressions:

N x	Denotes an integer x.
V v	Denotes a variable named v.
Plus a1 a2	Denotes the sum of two arithmetic expressions $a1$ and $a2$.

2.1.2 Boolean expressions

The language supports the following boolean expressions:

Bc True, Bc False	Denotes boolean constants TRUE and FALSE.
Not b	Denotes the logical negation of a boolean value b.
And b1 b2	Denotes the logical conjunction of two boolean values b1 and b2.
Less a1 a2	Denotes the comparison of two arithmetic expressions a1 and a2.

2.1.3 Commands

Finally, the language supports the following commands:

Assign v x	Assigns the outcome of an arithmetic expression x to variable v.
Seq c1 c2	Denotes a program which first executes $c1$ and then $c2$.
If b c1 c2	Denotes a program which executes $c1$ if b evaluates to TRUE and $c2$ otherwise.
While b c	Executes c as long as b evaluates to TRUE.
SKIP	Denotes the empty command which does nothing.

Note that the empty program is not useful in its own but it can be used, for example, if we only want to consider one branch of a conditional and do nothing in the other case as in the following example:

If (Less (V ”x”) (N 3)) (Assign ”x” (N 3)) SKIP

2.2 Tasks

Implement an interpreter for the programming language by filling out the template `src/Interpreter.hs` as follows:

1. Create data types to model a program:
   - Create a data type AExp to denote arithmetic expressions.
   - Create a data type BExp to denote boolean expressions.
   - Create a data type Com to denote commands.
2. Create a function

   $$\text{aval} \;::\; \text{AExp} \to \text{State} \to \text{Val}$$

   to evaluate arithmetic expressions. For example,

   ```
   >aval (Plus (N 3) (V "x")) (fromList [("x",0)])
   >3
   ```

3. Create a function

   $$\text{bval} \;::\; \text{BExp} \to \text{State} \to \textbf{Bool}$$

   to evaluate boolean expressions. For example,

   ```
   >bval (Less (N 3) (V "x")) (fromList [("x",0)])
   >False
   ```

4. Create a function

   $$\text{eval} \;::\; \text{Com} \to \text{State} \to \text{State}$$

   to evaluate commands. For example,

```
>eval SKIP (fromList [])
>fromList []
>
>eval (Assign "x" (N 5)) (fromList [("x",0)])
>fromList [("x",5)]
>
>eval (Seq (Assign "x" (N 5)) (Assign "x" (N 6)))
>     (fromList [("x",0)])
>fromList [("x",6)]
>
>eval (If (Less (V "x") (N 5)) (Assign "x" (N 6)) (SKIP))
>     (fromList [("x",4)])
>fromList [("x",6)]
>
>eval (If (Less (V "x") (N 5)) (Assign "x" (N 6)) (SKIP))
>     (fromList [("x",10)])
>fromList [("x",10)]
>
>eval (While (Less (V "x") (N 5))
>     (Assign "x" (Plus (V "x") (N 1))))
>     (fromList [("x",0)])
>fromList [("x",5)]
```

2.3 Notes

Types Vname, State and Val are imported from module `src/Machine.hs` and do not need to be redefined.

3 A basic Compiler

For the following exercise you need to implement a compiler which creates a set of instructions for the low-level machine developed in exercise 1 from a program written in the high-level language developed in exercise 2.

3.1 Compiler Specification

The compiler should be implemented as a function which takes as input a high-level program and returns a sequence of instructions for the low-level machine to implement the effects of the program.

3.1.1 Assignments

An assignment should be compiled to a sequence of instructions which first calculates the result of the arithmetic expression and then stores it in the corresponding variable. For example, the assignment

Assign x a

should be compiled to the following sequence of instructions:

code for a	store a to x

3.1.2 Sequential Composition

Sequential composition just combines the effects of the corresponding commands. For example the program

Seq c1 c2

would be compiled into a low-level program which first executes the instructions for c1 and then the instructions for c2:

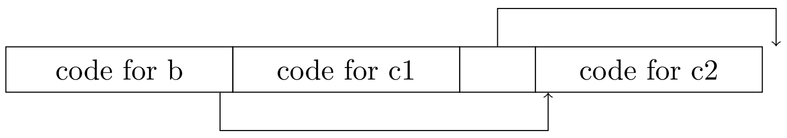
code for c1	code for c2

3.1.3 Conditionals

Programs involving conditionals and loops are more difficult and need the use of jump instructions. Consider, for example the program

If b c1 c2

This would require to evaluate the boolean expression b and then either execute c1 or c2. It would thus compile into the following sequence of instructions for the low-level machine (where arrows denote jump instructions):



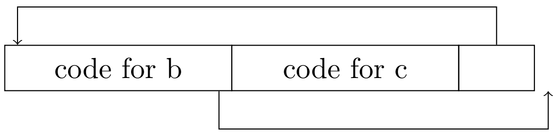
Thus, the code generated for the boolean expression b must contain a conditional jump instruction at the end, which jumps to the beginning of the code block for c2 if b evaluates to **False**. Moreover, another unconditional jump instruction is required to be added to the end of the code block for c1 to avoid execution of c2 after executing c1.

3.1.4 Loops

Finally, consider the program

While b DO c

This would require to evaluate the boolean expression b and then either execute the loop block or jump out of the loop:



Thus, again, the compiled code for boolean expression b needs to end with a conditional jump instruction to exit the loop if b evaluates to **False**. Moreover, the code for c needs to end with an unconditional jump expression to jump back to the beginning of the loop.

3.2 Tasks

Implement the compiler by filling out the template `src/Compiler.hs` as follows:

1. Define function

   acomp :: AExp → [Instr]

   to compile a simple arithmetic expression to a sequence of machine instructions. For example:

   ```
   >acomp (Plus (N 5) (V "x"))
   >[LOADI 5,LOAD "x",ADD]
   ```

2. Define function

   bcomp :: BExp → **Bool** → **Int** → [Instr]

   to compile a boolean expression to a sequence of machine instructions. Note that boolean expressions are only used in conditionals and loops and thus need to implement the corresponding jump instructions as explained above. The idea is that the second parameter contains the boolean value for which a jump is required and the third parameter contains the number of instructions to jump over. For example:

```
>bcomp (Bc True) True 3
>[JMP 3]
>
>bcomp (Bc False) False 3
>[JMP 3]
>
>bcomp (Bc True) False 3
>[]
>
>bcomp (Not (Bc False)) False 3
>[]
```

```
>bcomp (And (Bc True) (Bc False)) True 3
>[]
>
>bcomp (And (Bc False) (Bc True)) True 3
>[JMP 1,JMP 3]
>
>bcomp (And (Bc True) (Bc False)) False 3
>[JMP 3]
>
>bcomp (And (Bc False) (Bc True)) False 3
>[JMP 3]
```

```
>bcomp (Less (V "x") (N 5)) True 3
>[LOAD "x",LOADI 5,JMPLESS 3]
>
>bcomp (And (Less (V "x") (N 5)) (Bc True)) False 3
>[LOAD "x",LOADI 5,JMPGE 3]
>
>bcomp (And (Bc False) (Less (V "x") (N 5))) True 3
>[JMP 3,LOAD "x",LOADI 5,JMPLESS 3]
>
>bcomp (And (Bc False) (Less (V "x") (N 5))) False 3
>[JMP 6,LOAD "x",LOADI 5,JMPGE 3]
```

3. Use functions acomp and bcomp to define function

$$\text{ccomp} :: \text{Com} \rightarrow [\text{Instr}]$$

   to compile a high-level program to a sequence of machine instructions. For example:

```
>ccomp (If (Less (V "u") (N 1))
>        (Assign "u" (Plus (V "u") (N 1)))
>        (Assign "v" (V "u")))
>[LOAD "u",LOADI 1,JMPGE 5,LOAD "u",LOADI 1,
>        ADD,STORE "u",JMP 2,LOAD "u",STORE "v"]
>
>ccomp (While (Less (V "u") (N 1))
>        (Assign "u" (Plus (V "u") (N 1))))
>[LOAD "u",LOADI 1,JMPGE 5,LOAD "u",LOADI 1,
>        ADD,STORE "u",JMP (-8)]
```

4. Implement function main in `app/Main.hs` to allow to execute the compiler from command line. For example:

```
>stack exec coursework-exe "Assign␣\"x\"␣(Plus␣(N␣5)␣(N␣3))"
>[LOADI 5,LOADI 3,ADD,STORE "x"]
```

4 Verifying Compiler Correctness

For this exercise you need to verify the correctness of the compilation of arithmetic expressions.

4.1 Compiler Correctness

To verify correctness of your compiler for arithmetic expressions, you need to show that compilation preserves the meaning of arithmetic expressions. In other words, for every expression, the effect of evaluating the expression is the same as first compiling it into a sequence of machine instructions and then executing them on the low-level machine:

$$\begin{tikzcd}
\boxed{\text{expression}} \arrow[r, "\text{eval}"] \arrow[d, "\text{ccomp}"'] & \boxed{\text{value}} \arrow[d, leftrightarrow] \\
\boxed{\text{instructions}} \arrow[r, "\text{exec}"] & \boxed{\text{configuration}}
\end{tikzcd}$$

4.2 Tasks

Verify correctness of compilation for arithmetic expression by proving the following property in template `src/Verification.txt`:

exec (acomp a) (0,s ,[]) == (**length** (acomp a),s,[aval a s])

4.3 Notes

Use the notation discussed in the lecture to state the proof.