MP 1 – Formalizing a Simple Imperative Programming Language in Isabelle

CS 422 – Spring 2017

Revision 1.0

Assigned February 1, 2017
Due February 8, 2017, 8:00pm
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Turn-In Procedure

Put your code as plain text for this MP in a file named `mp1.thy`, and submit your plain text file `mp1.thy` by first adding it to your svn repository directory `assignments/mp1`, which may be done using the command (`svn add mp1.thy`) and then committing it using (`svn commit -m "submitting mp1" mp1.thy`). Your file should contain your name, and netid in a comment at the top, and it should contain your solution. It should be named `mp1.thy` and committed in your `assignments/mp1` directory.

3 Objectives

The purpose of this MP is to familiarize you with using Isabelle to specify a simple imperative programming language abstract syntax and Natural Semantics.

4 Background

In class, we have looked at a simple imperative programming language SIMPL1 and how to specify it in Natural Semantics, and how to formalize this Isabelle. In this assignment you will be asked to extend this Isabelle formalization for expressions and commands.

4.1 Syntax of SIMPL1

In class, we worked with specifying in Isabelle the language SIMPL1 whose concrete syntax is given by the BNF Grammar below:

$$
\begin{array}{lcl}
I & \in & \textit{Identifiers} \\
N & \in & \textit{Numerals} \\
E & ::= & N \mid I \mid E + E \mid E * E \mid E - E \\
B & ::= & \mathsf{true} \mid \mathsf{false} \mid B \& B \mid B \ \mathsf{or} \ B \mid \mathsf{not} \ B \mid E < E \mid E = E \\
C & ::= & \mathsf{skip} \mid C; C \mid \{C\} \mid I := E \mid \mathsf{if} \ B \ \mathsf{then} \ C \ \mathsf{else} \ C \ \mathsf{fi} \mid \mathsf{while} \ B \ \mathsf{do} \ C \ \mathsf{od}
\end{array}
$$

4.2 Natural Semantics for SIMPL1

Assuming a set Values of final results of expressions (in this case you can assume integers), and $m,\ m'$: Identifiers $\rightharpoonup$ Values, recall the Natural Semantics we gave for the SIMPL1 as follows:

Constants:

Identifiers: $(I, m) \Downarrow m(I)$ if $m(I)$ exists Numerals are values: $(N, m) \Downarrow N$

Booleans: $(\mathsf{true}, m) \Downarrow \mathsf{true}$ $(\mathsf{false}, m) \Downarrow \mathsf{false}$

Arithmetic Expressions:

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \oplus V = N}{(E \oplus E', m) \Downarrow N} \text{ where } \oplus \in \{+, *, -\} \text{ and } U, V \in \texttt{Values}$$

Arithmetic Relations:

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b}{(E \sim E', m) \Downarrow b} \text{ where } \sim \in \{==, <\}$$

Boolean Expressions:

$$\frac{(B, m) \Downarrow \mathsf{false}}{(B \& B', m) \Downarrow \mathsf{false}} \qquad \frac{(B, m) \Downarrow \mathsf{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b} \qquad \frac{(B, m) \Downarrow \mathsf{false}}{(\text{not } B, m) \Downarrow \mathsf{true}}$$

$$\frac{(B, m) \Downarrow \mathsf{true} \quad (B', m) \Downarrow b}{(B \& B', m) \Downarrow b} \qquad \frac{(B, m) \Downarrow \mathsf{true}}{(B \text{ or } B', m) \Downarrow \mathsf{true}} \qquad \frac{(B, m) \Downarrow \mathsf{true}}{(\text{not } B, m) \Downarrow \mathsf{false}}$$

Commands:

Assignment: $$\frac{(E, m) \Downarrow V}{(I := E, m) \Downarrow m[I \leftarrow V]} \text{ where } m[I \leftarrow V](J) = \begin{cases} V & \text{if } J = I \\ m(J) & \text{otherwise} \end{cases}$$

Skip: $(\mathsf{skip}, m) \Downarrow m$ Sequencing: $\dfrac{(C, m) \Downarrow m' \quad (C', m') \Downarrow m''}{(C; C', m) \Downarrow m''}$ Block: $\dfrac{(C, m) \Downarrow m'}{(\{C\}, m) \Downarrow m'}$

If-true: $\dfrac{(B, m) \Downarrow \mathsf{true} \quad (C, m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$ If-false: $\dfrac{(B, m) \Downarrow \mathsf{false} \quad (C', m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$

While-false: $$\frac{(B, m) \Downarrow \mathsf{false}}{(\text{while } B \text{ do } C \text{ od }, m) \Downarrow m}$$

While-true: $$\frac{(B, m) \Downarrow \mathsf{true} \ \ (C, m) \Downarrow m' \ \ (\text{while } B \text{ do } C \text{ od }, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od }, m) \Downarrow m''}$$

5 Problems

Your work for the problems below should be entered in the file `mp1.thy`, modifying definitions you find there to include the extensions described below.

1. (3 pts) Extend the `datatype` definition of `exp` to include a term constructor named `Div` for integer division. It should take two arguments of type `exp`. After you have successfully done this, you should be able to enter

```
term "Div (Plus (Val 3) (Val 4)) (Minus (Div (Val 6) (Val 2)) (Val 1))"
```

and you should see

"Div (Val 3 $+_E$ Val 4) (Div (Val 6) (Val 2) $-_E$ Val 1)" :: "exp"

with `Div` appearing in black (not blue) in all occurrences. The `term` line is also found in the file `mp1_tests.thy`. If you would like to use infix notation for `Div`, you may add `(infixl "\<div>\<^sub>E" 165)` to the end of the clause for `Div` in the `exp` datatype. If you do this and then put the cursor on the `term` line given above in `mp1_tests.thy`, you should see

"(Val 3 $+_E$ Val 4) $\div_E$ (Val 6 $\div_E$ Val 2 $-_E$ Val 1)" :: "exp"

This time, there should be no appearances of `Div`, but two infixed appearances of $\div_E$ instead.

2. (10 pts) Extend the semantics of `exp` to include a rule for the evaluation of division. The evaluation semantics of `Div x y` is the result of evaluating `x` to, say, u, evaluating `y` to, say, v, and if v is not 0, then evaluating u/v as the final result. If you have entered your rule correctly, the following theorem (in `mp1_tests.thy`) should be provable:

```
lemma test4:
 "eval_exp (Div (Val 2) (Var ''x''), Map.empty(''x'' := Some 13)) 0"
  by force
```

3. (6 pts) Extend the `datatype` definition of `commands` to include the term constructor named `RepeatCom`, which takes a `bool_exp` argument and a `command` argument. Upon successful completion of this, you should to able to enter

```
term "RepeatCom (AssignCom ''a'' (Var ''b'')) (Bool True)"
```

and see

"RepeatCom (''a'' ::= Var ''b'') (Bool True)" ::" command"

If you would like to have your abstract syntax displayed with something more like ordinary programming syntax, add `("REPEAT _/ UNTIL _/ DONE" [70,70] 70)` to the end of you clause for `RepeatCom`. Entering the above `term` line then display the following results:

"REPEAT ''a'' ::= Var ''b'' UNTIL Bool True DONE" ::" command"

4. (12 pts) To evaluate `RepeatCom` C B in a memory m, first evaluate C in m and use the resulting memory m' to evaluate B. If B evaluates to true then the result of evaluating `RepeatCom` C B in m in m'. If the result of B is false, then the result of evaluating `RepeatCom` C B in m is the result of evaluating `RepeatCom` C B in m'. If you have entered your rule correctly, the following theorem (in `mp1_tests.thy`) should be provable:

```
lemma test6:
  "eval_command
   (RepeatCom (AssignCom ''a'' (Var ''b'')) (Bool True),
    Map.empty(''b'' := Some 3))
   ((Map.empty(''b'' := Some 3))(''a'' := Some 3))"
  by force
```