RADBOUD UNIVERSITEIT NIJMEGEN

Coq opdracht

Suzanne van den Bosch
Studentnummer: s4021444

January 30, 2015

3 EXPRESSION COMPILER

Formalize both an interpreter and a compiler for a simple language of arithmetical expressions, and show that both give the same results. Compile the expressions to code for a simple stack machine. Use dependent types to make Coq aware of the fact that the compiled code will never lead to a run time error.

This is a short report that describes the formalization and discusses the choices made while formalizing.

3.1 INDUCTIVE DEFINITION OF EXP

The first assignment is to define the expressions. An expression is either a literal or two expressions connected with an operator. Because the literals are natural numbers, I decided to only use the operators + and $*$.
To avoid repetition of code, I decided to add a type '$Binop$', instead of defining both operators separately.

3.2 DEFINING EVAL

In this assignment we have to define the function $eval$, which has to give semantics to the expressions. Defining $eval$ was pretty straightforward, as I just used the semantics of + and $*$ on the natural numbers.

3.3 Inductive definition of RPN

Reverse Polish Notation doesn't use infix notation like we are used to, but it uses the so-called postfix notation. This means that all the operators follow it's operands. Reverse Polish Notation is therefore a string of operands and operators. We can also see this as a list of operands and operators, which is how I formalized it. The list is made up of 'RPN expressions'.

3.4 Defining compiler rpn

This assignment was pretty straightforward.

3.5 Defining rpn_eval

This assignment was pretty straightforward.

3.6 Proving theorem

$$\forall\, e : Exp,\ Some\,(eval\ e) = rpn\_eval\,(rpn\ e)$$

Proving this theorem I got stuck almost immediately. But then Robbert gave me the hint that I had to use more Theorem's. So I used the following theorem to help me:

$$\forall e : Exp, \forall s : listnat, \forall t : RPN, rpn\_eval1\ s\ ((rpn\ e) +\!+ t) = rpn\_eval1(eval\ e :: s)\ t$$

First I just "Admitted" that theorem, and then the proof of the original theorem became very easy. And it turned out the second theorem was the most difficult to prove. I used some asserts, I decided not to make theorems out of the asserts, because it was easier to proof the asserts in the theorem, than start over in a new theorem.

3.7 Including variables

For this assignment I first copy-pasted what I already had. Then I redefined all expressions to include a variable. I decided to store the variables in a list and defined a 'lookup' function. This function is used when the variable is used. The variable is a function which takes a natural number as input and outputs expression. $Var0$ corresponds to the first index of the variable list.
Here the proof of the theorem itself was almost as easy as without variables, because not much changed. However, the "help" theorem was a lot more difficult and I eventually got stuck in the next situation:
Assumption:

```
IHt : Some (lookup_var n varList) = rpn_evalv1 (lookup_var n varList :: l) t varList
```

Goal:

```
rpn_evalv1 (lookup_var n varList :: l) t varList =
rpn_evalv1 (lookup_var n varList :: l) (a :: t) varList
```

I just don't see how I can make these two parts equal.

3.8 AVOIDING “NONE" TERMS

You could add an extra term, for example “Nonexist" while defining expressions. This way you can avoid the term None while defining rpn_eval. And then you would have to omit the Option entirely, because if you don’t use the None, then there’s no point in using an Option.