Verification with Proof Score of Arithmetic Expression Compiler

Lecture Note 6

Topics

- Correctness of a compiler for arithmetic expressions
- Formalization of the correctness
- Proof by induction on the structure of expressions (i.e. structural induction)
- Verification with proof score of an arithmetic expression complier

Correctness of Compilers

- One possible definition that a compiler is correct:

  For any program p, the result of executing p by an interpreter is the same as that of producing an instruction sequence by the compiler for p and then executing the instruction sequence by a virtual machine.
- We will verify that a compiler for arithmetic expressions is correct.

Natural Numbers with the "`errNat`" -- signature

Module `NATerr`:

```
mod! NATerr {
  pr(NAT)
  [Nat ErrNat < Nat&Err]
  -- For verification
  op errNat : -> ErrNat {constr}
  --
  op _=_ : Nat&Err Nat&Err -> Bool {comm}
  op _+_ : Nat&Err Nat&Err -> Nat&Err {comm}
  op _*_ : Nat&Err Nat&Err -> Nat&Err {comm}
  op sd : Nat&Err Nat&Err -> Nat&Err {comm}
  op _quo_ : Nat Zero -> ErrNat
  op _quo_ : Nat&Err Nat&Err -> Nat&Err
```

Natural Numbers with the "`errNat`" -- equations defining errors also

```
vars M N : Nat
var NE : Nat&Err
-- _=_
eq (NE = NE) = true .
eq (N = M) = (N == M) .
eq (errNat = N) = false .
-- _+_
eq NE + errNat = errNat .
```

```
-- _*_
eq NE * errNat = errNat .
-- sd
eq sd(NE,errNat) = errNat .
-- quo
eq M quo 0 = errNat .
eq NE quo errNat = errNat .
eq errNat quo NE = errNat .
}
```


Expressions

- Module EXP:

```
mod! EXP {
 pr(NATerr)
 [Nat&Err < Exp]
 op _++_ : Exp Exp -> Exp
          {constr l-assoc prec: 30}
 op _--_ : Exp Exp -> Exp
          {constr l-assoc prec: 30}
 op _**_ : Exp Exp -> Exp
          {constr l-assoc prec: 29}
 op _//_ : Exp Exp -> Exp
          {constr l-assoc prec: 29}
}
```

Interpreter

Function interpret:

```
op interpret : Exp -> Nat&Err
eq interpret(N) = N .
eq interpret(E1 ++ E2)
     = interpret(E1) + interpret(E2) .
eq interpret(E1 -- E2)
     = sd(interpret(E1),interpret(E2)) .
eq interpret(E1 ** E2)
     = interpret(E1) * interpret(E2) .
eq interpret(E1 // E2)
     = interpret(E1) quo interpret(E2) .
```


Instructions

Instructions

```
op push : Nat&Err-> Command {constr}
op multiply : -> Command {constr}
op divide : -> Command {constr}
op add : -> Command {constr}
op minus : -> Command {constr}
```

Virtual Machine (1)

```
vars N N1 N2 : Nat&Err
```

Function vm:

```
op vm : CList -> Nat&Err
eq vm(CL) = exec(CL,empstk) .
```

Function exec:

```
op exec : CList Stack -> Nat&Err
eq exec(nil,empstk) = errNat .
eq exec(nil,N | empstk) = N .
eq exec(nil,N | N1 | Stk) = errNat .
eq exec(push(N) | CL,Stk) = exec(CL,N | Stk) .
eq exec(add | CL,empstk) = errNat .
eq exec(add | CL,N | empstk) = errNat .
eq exec(add | CL,N2 | N1 | Stk)
     = exec(CL,N1 + N2 | Stk) .
```

The empty stack

An instruction sequence (list)

A stack implemented as a list of natural numbers & errNat.

Exceptions

Virtual Machine (2)

Function exec (continued):

```
eq exec(multiply | CL,empstk) = errNat
eq exec(multiply | CL,N | empstk) = errNat .
eq exec(multiply | CL,N2 | N1 | Stk)
     = exec(CL,N1 * N2 | Stk) .
eq exec(divide | CL,empstk) = errNat
eq exec(divide | CL,N | empstk) = errNat .
eq exec(divide | CL,N2 | N1 | Stk)
     = exec(CL,N1 quo N2 | Stk) .
eq exec(minus | CL,empstk) = errNat .
eq exec(minus | CL,N | empstk) = errNat .
eq exec(minus | CL,N2 | N1 | Stk)
     = exec(CL,sd(N1,N2) | Stk) .
eq exec(CL,errNat | Stk) = errNat .
eq exec(CL,N | errNat | Stk) = errNat .
```

Exceptions

Compiler

Function `compile`:

```
op compile : Exp -> Clist
var N : Nat&Err
vars E E1 E2 : Exp
eq compile(N) = push(N) | nil .
eq compile(E1 ++ E2)
     = compile(E1) @ compile(E2) @ (add | nil) .
eq compile(E1 -- E2)
     = compile(E1) @ compile(E2) @ (minus | nil) .
eq compile(E1 ** E2)
     = compile(E1) @ compile(E2) @ (multiply | nil) .
eq compile(E1 // E2)
     = compile(E1) @ compile(E2) @ (divide | nil) .
```


Formalization of the Correctness (1)

- The interpreter can be regarded as the specification of the compiler.
- The correctness of the compiler can be defined as follows:
  - For any expression e, the result of executing e by the interpreter is the same as that of producing an instruction sequence by the compiler for e and then executing the instruction sequence by the virtual machine.
- This is formalized as follows:

  For all `E:Exp, interpret(E) = vm(compile(E))`

Formalization of the Correctness (2)

Module COMPILER-THEOREM:

```
mod* COMPILER-THEOREM {
  pr(INTERPRETER)
  pr(VM)
  pr(COMPILER)
-- theorem
  op th1 : Exp -> Bool
  eq th1(E:Exp) =
        (interpret(E) = vm(compile(E))) .
}
```


Proof by Induction on the Structure of Expressions

- For a function (predicate) p : Exp -> Bool, the following two formulas are equivalent:
  - (1) p(E) for all E:Exp
  - (2) p(N), p(E1) and p(E2) implies p(E1 ++ E2), …, p(E1) and p(E2) implies p(E1 // E2) for all N:Nat and E1,E2:Exp.
- Therefore, to prove (1), it suffices to show
  - (i) Base case: p(n) for an arbitrary n:Nat.
  - (ii) Induction case:
    1. p(e1 ++ e2) assuming p(e1),p(e2) for arbitrary e1,e1:Exp.
       …
    4. p(e1 // e2) assuming p(e1),p(e2) for arbitrary e1,e1:Exp.
- (i) is called the *base case*, and (ii) the *induction case*.
- p(e1) and p(e2) are called the *induction hypotheses*.

Verification of Compiler (1)

The proof passage of the base case is as follows:

```
open COMPILER-THEOREM
  op m : -> Nat&Err .
-- check
  red th1(m) .
close
```

CafeOBJ returns `true` for the proof passage.

Verification of Compiler (2)

The initial proof passage of the induction case where _++_ is taken into account:

```
open COMPILER-THEOREM
-- arbitrary values
  ops e1 e2 : -> Exp .
-- induction hypothesis
  eq interpret(e1) = vm(compile(e1)) .
  eq interpret(e2) = vm(compile(e2)) .
-- check
  red th1(e1 ++ e2) .
```

CafeOBJ returns the following:

```
(exec(compile(e1),empstk) + exec(compile(e2),empstk))
 = exec(compile(e1) @ compile(e2) @ (add | nil),empstk)
```

Verification of Compiler (3)

One possible way to discharge the case is to conjecture the following lemma and use it in the proof passage:

For any `E1,E2:Exp`,

```
(exec(compile(E1),empstk) + exec(compile(E2),empstk))
 = exec(compile(E1) @ compile(E2) @ (add | nil),empstk)
```

But, this lemma seems too specific to the case.
It seems more preferable to conjecture a more general one.
Then, let us focus on the RHS of the lemma (or the result returned by CafeOBJ).
A more general term of the RHS is

```
exec(compile(E) @ L,S)
```


Verification of Compiler (4)

When the vm executes `compile(E)`, the term is most likely to be

```
exec(L,vm(compile(E)) | S)
```

So, one possible lemma:

For any `E:Exp, L:CList, S:Stack,`

```
exec(compile(E) @ L,S) = exec(L,vm(compile(E)) | S)
```

Then, the followings are added to `COMPILER-THEORM` to get the new module `COMPILER-THEOREM-with-LEMMA`

```
-- lemma
op lem1 : Exp CList Stack -> Bool
eq lem1(E:Exp,L:CList,S:Stack)
   = (exec(compile(E) @ L,S) = exec(L,vm(compile(E)) | S)) .
```

Verification of Compiler (5)

The proof passage becomes:

```
open COMPILER-THEOREM-with-LEMMA
-- arbitrary values
  ops e1 e2 : -> Exp .
-- lemmas
  eq exec(compile(E:Exp) @ L:CList,S:Stack)
     = exec(L,vm(compile(E)) | S) . -- lem1(E,L,S)
-- induction hypothesis
  eq interpret(e1) = vm(compile(e1)) .
  eq interpret(e2) = vm(compile(e2)) .
-- check
  red th1(e1 ++ e2) .
close
```


Exercises

1. Write the proof score for verifying the lemma:
   `lem1(E:Exp,L:Clist,S:Stack).`