Specification and Verification of Arithmetic Expression Compiler

Lecture Note 04
Formal Methods (i613-0912)

Topics

- Specification of an interpreter, a virtual machine, and a compiler
- Verification of the compiler
  - How to split a case into multiple sub-cases
  - How to conjecture/use lemmas

Expressions

- Expressions treated here are composed of natural numbers, addition, subtraction, multiplication, and division.
- Inductively defined as follows:
  1. Natural numbers are expressions.
  2. If E_1 and E_2 are expressions, so are
     - ✓ E_1 ++ E_2 (Addition)
     - ✓ E_1 –– E_2 (Subtraction; absol value of the diff)
     - ✓ E_1 ** E_2 (Multiplication)
     - ✓ E_1 // E_2 (Division)

Natural Numbers in CafeOBJ (1)

- Module `PNAT`:

```
mod! PNAT principal-sort ErrorNat {
  [Zero NzNat < Nat < ErrorNat]
  op 0 : -> Zero {constr}
  op s : Nat -> NzNat {constr}
  op errorNat : -> ErrorNat {constr}
  op _=_ : ErrorNat ErrorNat -> Bool {comm}
  op _<_ : ErrorNat ErrorNat -> Bool
  op _+_ : Nat Nat -> Nat
  op _+_ : ErrorNat ErrorNat -> ErrorNat
  op _*_ : Nat Nat -> Nat
  op _*_ : ErrorNat ErrorNat -> ErrorNat
  op sd : Nat Nat -> Nat
  op sd : ErrorNat ErrorNat -> ErrorNat
  op _quo_ : Nat Zero -> ErrorNat
  op _quo_ : Nat NzNat -> Nat
  op _quo_ : ErrorNat ErrorNat -> ErrorNat
  …
```

for exceptions such as “divided by zero”

denotes exceptions

Natural Numbers in CafeOBJ (2)

- In addition to the standard equations of `_=_`,

```
eq (errorNat = N) = false .
eq (N = errorNat) = false .
eq (errorNat = errorNat) = true .
```

- In addition to the standard equations of `_+_`,

```
eq M + errorNat = errorNat .
eq errorNat + N = errorNat .
eq errorNat + errorNat = errorNat .
```

- Definition of `_quo_`:

```
eq M quo 0 = errorNat .
eq M quo s(N)
   = (if M < s(N) then 0 else s(sd(M,s(N)) quo s(N)) fi) .
eq M quo errorNat = errorNat .
eq errorNat quo N = errorNat .
eq errorNat quo errorNat = errorNat .
```

`M` and `N` are variables of `Nat`.

Expressions in CafeOBJ

- Module `EXP`:

```
mod! EXP {
  pr(PNAT)
  [Nat < 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}
}
```

✓ `l-assoc` says that `e1 ++ e2 ++ e3` is parsed as `(e1 ++ e2) ++ e3`.

✓ `prec:` n specifies precedence of operators; the less n is, the higher the precedence is. Hence, `e1 ++ e2 ** e3` is parsed as `e1 ++ (e2 ** e3)`.

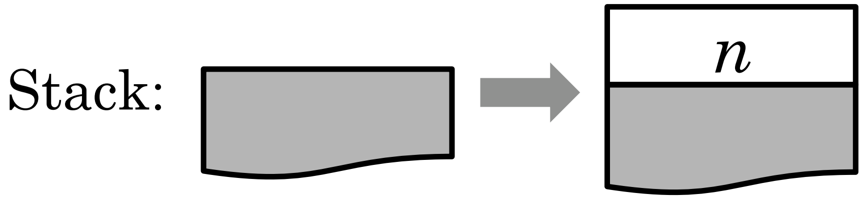
An Interpreter in CafeOBJ

- It takes an expression and returns the result (a natural number or `errorNat`) obtained by calculating the expression.

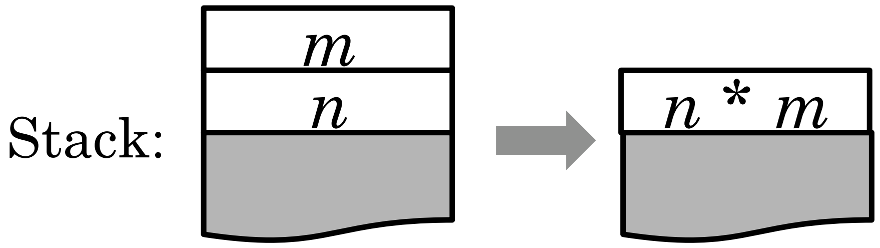
```
op interpret : Exp -> ErrorNat
eq interpret(N) = N .
eq interpret(E1 ++ E2)
   = interpret(E1) + interpret(E2) .
eq interpret(E1 -- E2)
   = sd(interpret(E1),interpret(E2)) .
eq interpret(E1 ** E2)
   = interpret(E1) * interpret(E2) .
eq interpret(E1 // E2)
   = interpret(E1) quo interpret(E2) .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

A Virtual Machine (1)

- A stack machine with five instructions: push(n), multiply, divide, add, and minus.
  - push(n):

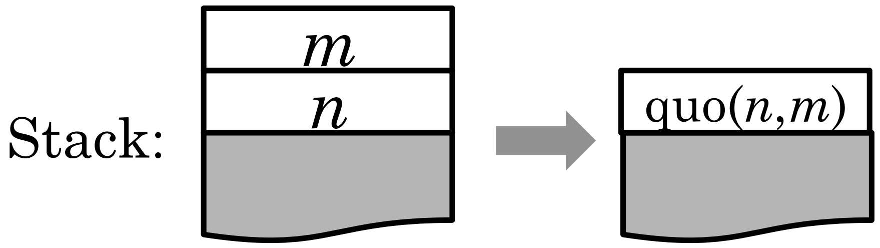


  - multiply:

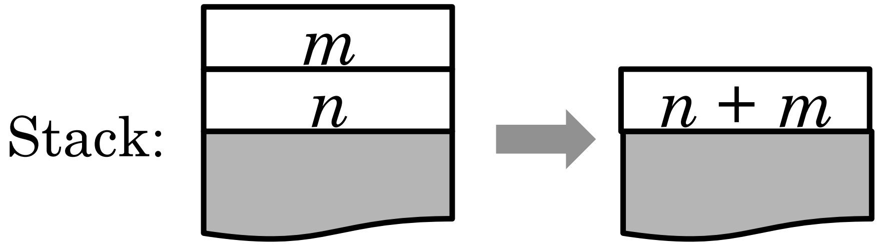


A Virtual Machine (2)

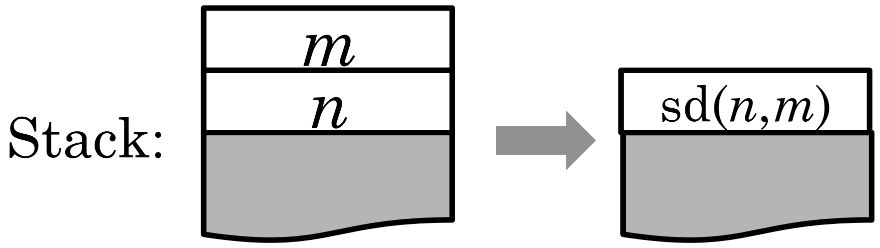
- divide:



- add:



- minus:



Instructions in CafeOBJ

- Module `COMMAND`:

```
mod! COMMAND principal-sort Command {
  pr(PNAT)
  [Command]
  op push : Nat -> Command {constr}
  op multiply : -> Command {constr}
  op divide : -> Command {constr}
  op add : -> Command {constr}
  op minus : -> Command {constr}
}
```

Generic Lists

- Module `LIST`:

```
mod! LIST (M :: TRIV) {
  [List]
  op nil : -> List {constr}
  op _|_ : Elt.M List -> List {constr}
  op _@_ : List List -> List {assoc}
  var E : Elt.M
  vars L1 L2 : List
  eq nil @ L2 = L2 .
  eq (E | L1) @ L2 = E | (L1 @ L2) .
  eq L1 @ nil = L1 .
}
```

✓ You can prove that `_@_` is associative and `nil` is a right identity of `_@_` from the two equations.

Instruction Sequences & Stacks

- Instruction sequences represented as lists of commands:

```
mod! CLIST {
  pr(LIST(COMMAND) * {sort List -> CList})
}
```

- Stacks represented as lists of natural numbers and `errorNat`:

```
mod! STACK {
  pr(LIST(PNAT)
     * {sort List -> Stack, op nil -> empstk})
}
```

A Virtual Machine in CafeOBJ (1)

- Operator `vm`:

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

- Operator `exec`:

```
op exec : CList Stack -> ErrorNat
```

✓ When the instruction sequence becomes empty (nil), if the stack contains only one natural number, then the virtual machine can terminate successfully.

```
eq exec(nil,N | empstk) = N .
```

✓ Otherwise, something wrong must have happened.

```
eq exec(nil,empstk) = errorNat .
eq exec(nil,N | N1 | Stk) = errorNat .
```

A Virtual Machine in CafeOBJ (2)

- Operator `exec` (cont):
  - ✓ When the top of the instruction sequence is `push(N)`, `N` is put onto the stack.

```
eq exec(push(N) | CL,Stk) = exec(CL,N | Stk) .
```

  - ✓ When the top of the instruction sequence is one of the remaining four instructions, say `add`, if the stack contains at least two natural numbers, then the virtual machine can execute the instruction successfully.

```
eq exec(add | CL,N2 | N1 | Stk)
     = exec(CL,N1 + N2 | Stk) .
```

  - ✓ Otherwise, an exception is raised.

```
eq exec(add | CL,empstk) = errorNat .
eq exec(add | CL,N | empstk) = errorNat .
```

A Virtual Machine in CafeOBJ (3)

- Operator `exec` (cont):
  - ✓ Since the stack may contain `errorNat`, we need to have two more equations to take into account all cases.

```
eq exec(CL,errorNat | Stk) = errorNat .
eq exec(CL,N | errorNat | Stk) = errorNat .
```

A Compiler in CafeOBJ

- It takes an expression and generates an instruction sequence.

```
op compile : Exp -> Clist
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) .
```

Correctness of the Compiler

- The interpreter is used as an oracle.
- The correctness of the compiler is defined as follows:
  - *If the interpreter returns a natural number as the result of executing an expression, then the compiler generates an instruction sequence for the expression and the virtual machine returns for the instruction sequence the same natural number as the result returned by the interpreter.*
- This is formalized as follows:

```
op th1 : Exp Nat -> Bool
eq th1(E,N)
   = (interpret(E) = N implies vm(compile(E)) = N) .
```

These, together with some constants such as `e` and `n` denoting an arbitrary expression and an arbitrary natural number, are declared in module `THEOREM-COMPILER`

Verification of Compiler (1)

- `th1(E,N)` is proved by induction on `E`, namely by structural induction on expressions.
- Base case:

```
open THEOREM-COMPILER
-- check
  red th1(m,n) .
close
```

✓ `m` and `n` are constants denoting arbitrary natural numbers.

- Induction case: There are 4 sub-cases.

```
1. e1 ++ e2
2. e1 -- e2
3. e1 ** e2
4. e1 // e2
```

Verification of Compiler (2)

♦ Induction case (`e1 ++ e2`):

```
open THEOREM-COMPILER
-- check
  red th1(e1 ++ e2,n) .
close
```

✓ CafeOBJ returns neither `true` nor `false`.
✓ Since `interpret(e1)` and `interpret(e2)` appear in the result, and they return `errorNat` or a natural number, then we split the case into 4 sub-cases based on the values returns by `interpret(e1)` and `interpret(e2)`.

Verification of Compiler (3)

- The 4 sub-cases are as follows:

	`interpret(e1)`	`Interpret(e2)`
case 1	`errorNat`	`errorNat`
case 2	a natural number `k1`	`errorNat`
case 3	`errorNat`	a natural number `k2`
case 4	a natural number `k1`	a natural number `k2`

✓ CafeOBJ returns true for the first 3 cases, but neither `true` nor `false` for the last one.
✓ Since `k1 + k2 = n` appers in the result for case 4 and then case 4 is split into two sub-cases.

	`k1 + k2 = n`
case 4-1	`false`
case 4-2	`true`

✓ CafeOBJ returns true for case 4-1, but not for case 4-2.

Verification of Compiler (4)

♦ CafeOJB returns the following term for case 4-2.

```
exec(compile(e1) @ compile(e2) @ (add | nil),            ... (1)
     empstk) = n
```

✓ You can do further case splitting, but it does not seem to make a very meaningful progress.
✓ The induction hypotheses `exec(compile(e1),empstk) = K1` and `exec(compile(e2),empstk) = K2` do not seem to help.
✓ Then, try to conjecture a lemma.
✓ Imagine how the LHS of (1) changes if it is successfully computed.

```
exec(compile(e2) @ (add | nil),                          ... (2)
     intepret(e1) | empstk) = n

exec(add | nill,                                         ... (3)
     interpret(e2) | interpret(e1) | empstk) = n
```

Verification of Compiler (5)

- It seems true that (2) implies (1) and (3) implies (2), letting us come up with the lemma:

```
op th2 : Exp CList Stack Nat -> Bool
eq th2(E,L,S,N) = (exec(L,interpret(E) | S) = N
                   implies exec(compile(E) @ L,S) = N) .
```

  - ✓ The conjecture says that
    - If (2) is assumed, then an instance of the lemma is the exactly the same as (1), discharging case 4-2.
    - If the negation of (2) is assumed, then an instance of the lemma becomes false because of `k1 + k2 = n`, discharging case 4-2.
  - ✓ Then, case 4-2 is split into two sub-cases based on (2): case 4-2-1 for `true` and case 4-2-2 for `false`.

```
exec(compile(e1) @ compile(e2) @ (add | nil),empstk) = n              ... (1)
exec(compile(e2) @ (add | nil),intepret(e1) | empstk) = n             ... (2)
exec(add | nill,interpret(e2) | interpret(e1) | empstk) = n           ... (3)
```

Verification of Compiler (6)

- The proof passage of case 4-2-1 is as follows:

(2)

(2) Implies (1)

```
open THEOREM-COMPILER
-- arbitrary values
  ops k1 k2 : -> Nat .
-- assumptions
  eq interpret(e1) = k1 .
  eq interpret(e2) = k2 .
  eq k1 + k2 = n .
  eq exec((compile(e2) @ (add | nil)),
          (k1 | empstk)) = n .
-- check
  red th2(e1,compile(e2) @ (add | nil),empstk,n)
      implies th1(e1 ++ e2,n) .
close
```

(1)

Verification of Compiler (7)

- The proof passage of case 4-2-2 is as follows:

```
open THEOREM-COMPILER
-- arbitrary values
  ops k1 k2 : -> Nat .
-- assumptions
  eq interpret(e1) = k1 .
  eq interpret(e2) = k2 .
  eq k1 + k2 = n .
  eq (exec((compile(e2) @ (add | nil)),
           (k1 | empstk)) = n)  = false .
-- check
  red th2(e2,add | nil,k1 | empstk,n)
      implies th1(e1 ++ e2,n) .
close
```

(2)

(2)

(3) Implies (2)

Verification of Compiler (8)

- The remaining three sub-cases of the induction case can be proved likewise.

Proof of the Lemma (1)

- `th2(E,L,S,N)` is proved by induction on `E`.
- Base case can be straightforwardly proved.
- Induction case consists of 4 sub-cases.
- Induction case (`e1 ++ e2`):

```
open THEOREM-COMPILER
-- check
  red th2(e1 ++ e2,l,s,n) .
close
```

✓ `e1`, `e2`: arb exps
✓ `l`: an arb instr seq
✓ `s`: an arb stack
✓ `n`: an arb natural num

✓ CafeOBJ returns neither `true` nor `false`.
✓ Since `interpret(e1)` and `interpret(e2)` appear in the result, and they return errorNat or a natural number, then we split the case into 4 sub-cases based on the values returns by `interpret(e1)` and `interpret(e2)`.

Proof of the Lemma (2)

- The 4 sub-cases are as follows:

	`interpret(e1)`	`Interpret(e2)`
case 1	`errorNat`	`errorNat`
case 2	a natural number `k1`	`errorNat`
case 3	`errorNat`	a natural number `k2`
case 4	a natural number `k1`	a natural number `k2`

✓ CafeOBJ returns true for the first 3 cases, but neither `true` nor `false` for the last one.
✓ Since `exec(l,((k1 + k2) | s)) = n` appers in the result for case 4 and then case 4 is split into two sub-cases.

	`exec(l,((k1 + k2)	s)) = n`
case 4-1	`false`
case 4-2	`true`

✓ CafeOBJ returns true for case 4-1, but not for case 4-2.

Proof of the Lemma (3)

♦ CafeOJB returns the following term for case 4-2.

```
exec(compile(e1) @ compile(e2) @ (add | l),s) = n
```

✓ This lets us think that the following instance of the induction hypothesis can be used:

```
th2(e1,compile(e2) @ (add | l),s,n)
```

which is equivalent to

```
exec(compile(e2) @ (add | l),interpret(e1) | s) = n
implies
exec(compile(e1) @ compile(e2) @ (add | l),s) = n
```

✓ Then, case 4-2 is split into two sub-cases based on the premise of the instance.
✓ CafeOBJ returns `true` for both sub-cases.

Proof of the Lemma (4)

- The remaining three sub-cases of the induction case can be proved likewise.

Exercises

- Complete writing the proof scores.
- `th2(E,L,S,N)` is a generalization of `th1(E,N)` and then the latter can be derived from the former, which allows you to write a simpler proof score of `th1(E,N)`. Write a simpler proof score of `th1(E,N)`.