Introduction to Specification & Verification in CafeOBJ
2. An Arithmetic Expression Compiler

Kazuhiro Ogata
(JAIST)

Gist

• Some more concepts such as order sorts and parameterized modules on CafeOBJ are given.
• It is proved that an arithmetic expression compiler is correct with respect to an expression interpreter by writing proof scores in CafeOBJ.
Roadmap

• Some more concepts on CafeOBJ
  – order sorts, parameterized modules, renaming
  – tight semantics & loose semantics
• Proofs of some properties on generic lists
• A simple calculator
  – Two versions of a calculator
    1) interpreter
    2) compiler + virtual machine
  – Verification of the compiler

Order Sorts (1)

• Sorts can be partially ordered.
  – Sort Nat can be classified into two subsorts Zero and NzNat.
  – Sort Zero denotes the set \{0\}.
  – Sort NzNat denotes the set \{1, 2, ...\}.
• The sub-sort relation is declared as follows:
  \[ \text{Zero NzNat} < \text{Nat} \]
• We may have a super-sort ErrorNat of sort Nat to handle some exceptions.
  If that is the case, the sub-sort relation is as follows:
  \[ \text{Zero NzNat} < \text{Nat} < \text{ErrorNat} \]
• We have the following constant of sort ErrorNat:
  \( \text{op errorNat} : \rightarrow \text{ErrorNat} \)
Order Sorts (2)

• Let us consider an operator that returns the quotient obtained by dividing a natural number \(X\) by a natural number \(Y\).
• Only if \(Y\) is not zero, the quotient is well defined. If \(Y\) is zero, however, an exception occurs.
• To express the exception, the operator is declared as follows:
  
  \[
  \begin{align*}
  \text{op}_{\text{quo}} & : \text{Nat Zero} \rightarrow \text{ErrorNat} \\
  \text{op}_{\text{quo}} & : \text{Nat NzNat} \rightarrow \text{Nat} \\
  \text{op}_{\text{quo}} & : \text{ErrorNat ErrorNat} \rightarrow \text{ErrorNat}
  \end{align*}
  \]
• Operator \(\text{op}_{\text{quo}}\) is defined as follows:
  
  \[
  \begin{align*}
  \text{eq } M \text{ quo } 0 &= \text{errorNat} \\
  \text{eq } M \text{ quo } s(N) &= \begin{cases} 
  0 & \text{if } M < s(N) \\
  \text{s(sd}(M,s(N)) \text{ quo } s(N)) \text{ fi} & \text{else}
  \end{cases} \\
  \text{eq } M \text{ quo } \text{errorNat} &= \text{errorNat} \\
  \text{eq } \text{errorNat} \text{ quo } N &= \text{errorNat} \\
  \text{eq } \text{errorNat} \text{ quo } \text{errorNat} &= \text{errorNat}
  \end{align*}
  \]
  where \(M\) and \(N\) are variables of sort \(\text{Nat}\).

Some More Operators on \textbf{Nat}

• In addition to 0, s, \(+\), \(*\) and \(=\), the following operators are declared and defined in module \texttt{PNAT}:
  
  – \(\text{X < Y}\) checks if \(X\) is less than \(Y\).
  – \(\text{s d}(X, Y)\) computes the absolute value of the difference between \(X\) and \(Y\).
  – \(\text{X quo Y}\) computes the quotient obtained by dividing \(X\) by \(Y\).
Parameterized Modules (1)

• Modules can have parameters: *parameterized modules*.
• Parameterized modules make it possible to define *generic data structures* such as generic lists, which are lists of an arbitrary sort.
• Parameterized modules are *instantiated* to make modules for concrete data structures such as lists of natural numbers.

Parameterized Modules (2)

• Parameterized modules are declared as follows:

\[
\text{mod! } M(X_1 : : C_1, \ldots, X_n : : C_n) \{ \ldots \}
\]

where \( C_1, \ldots, C_n \) are modules, which constrain the formal parameters \( X_1, \ldots, X_n \).
Parameterized Modules (3)

- Parameterized modules are instantiated as follows:
  
  \[ M(X_1 \leq V_1, \ldots, X_n \leq V_n) \]
  
  where \( V_1, \ldots, V_n \) are views, which are as follows:

  \[
  \text{view } V_i \text{ from } C_i \text{ to } A_i \{ \\
  \quad \text{sort } \text{SortIn}C_i \rightarrow \text{SortIn}A_i, \\
  \quad \ldots \\
  \quad \text{op } \text{OpIn}C_i \rightarrow \text{OpIn}A_i \\
  \quad \ldots \}
  \]

  where \( A_1, \ldots, A_n \) are actual parameters, which are modules.

  Views correspond sorts and operators in \( C_i \) with those in \( A_i \).

  Sorts and operators in \( C_i \) are replaced with the correspondents in \( A_i \).

Generic Lists (1)

- A module for generic lists:

  \[
  \text{mod! LI}ST(M :: \text{TRIV}) \{ \\
  \quad \text{[List]} \\
  \quad \text{op } \text{nil} : \rightarrow \text{List} \\
  \quad \text{op } \text{_|_} : \text{Elt.M List} \rightarrow \text{List} \\
  \quad \text{op } \text{@} : \text{List List} \rightarrow \text{List} \\
  \quad \text{var } E : \text{Elt.M} \text{ vars L1 L2 : List} \\
  \quad \text{eq } \text{nil } \text{@} L2 = L2 . \\
  \quad \text{eq } (E | L1) \text{@} L2 = E | (L1 @ L2) .
  \}
  \]

  - \text{TRIV} is a built-in module where only one sort \text{Elt} is declared.
  - Sorts and functions \text{o} in formal parameters \text{P} are referred as \text{o . P}, say \text{Elt.M}.
  - \text{Elt.M} plays as an arbitrary sort, which can be replaced with a concrete sort by instantiating \text{LIST}.
  - Operator \text{@} concatenates two lists given as arguments.
Generic Lists (2)

- LIST is instantiated with PNAT as follows:
  
  mod! ERRNATLIST { pr(LIST(TRIV2PNAT)) }  

- TRIV2PNAT is the view declared as follows:
  
  view TRIV2PNAT from TRIV to PNAT
  { sort Elt -> ErrorNat }  

- The instance of module LIST specifies lists of natural numbers and errorNat.

- Examples
  
  - red in ERRNATLIST : nil .
  - red in ERRNATLIST : 0 | s(0) | s(s(0)) | nil .
  - red in ERRNATLIST : 0 | s(0) | errorNat | nil .

Renaming

- Renaming is to change the names of sorts and functions.

  Module * {sort SName -> AnotherSName,  
  ...
  op FName -> AnotherFName,  
  ...
  }

- Example

  mod! ERRNATLIST {
    pr(LIST(TRIV2PNAT) * {sort List -> ENList,  
    op nil -> empty,  
    op _|_ -> _||_})
  }

  red in ERRNATLIST : empty .
  red in ERRNATLIST : 0 || s(0) || s(s(0)) || empty .
  red in ERRNATLIST : 0 || s(0) || errorNat || empty .
Tight Semantics & Loose Semantics

• When a module has *tight semantics*, the module denotes the smallest algebra that satisfies what are described in the module.
  – Such a module is specified as \( \text{mod}! \ M \ \{ \ldots \} \).
  – Module PNAT denotes (the algebra that exactly corresponds to) natural numbers.
• When a module has *loose semantics*, the module denotes an arbitrary algebra that satisfies what are described in the module.
  – Such a module is specified as \( \text{mod}^* \ M \ \{ \ldots \} \).
  – Module TRIV denotes an arbitrary algebra.

Some Properties on \(_@_\)

• We formally prove the following properties:
  – \(_@_\) is associative.
  – nil is a right identity of \(_@_\).
• To this end,
  – we use EQTRIV instead of TRIV.
    \[
    \begin{align*}
    \text{mod}^* \ \text{EQTRIV} \ \{ \ \text{[Elt]} \} \quad \text{op } \_=\_ : \text{Elt Elt } &\rightarrow \text{Bool } \{ \text{comm} \} \\
    \text{var } E : \text{Elt} \\
    \text{eq } (E = E) &\rightarrow \text{true .} \\
    \end{align*}
    \]
  – The equality predicate on lists is declared and defined in module LIST.
Proof by Induction on Lists

• For an operator (predicate) \( p : \text{List} \rightarrow \text{Bool} \), the following two formulas are equivalent:
  
  (1) \( p(L) \) for all \( L: \text{List} \)

  (2) \( p(\text{nil}) \) and \( (p(L) \text{ implies } p(E \mid L)) \) for all \( E: \text{Elt.}X \) and all \( L: \text{List} \).

• Therefore, to prove (1), it suffices to show

  (i) \( p(\text{nil}) \)

  (ii) \( p(e \mid l) \) assuming \( p(l) \) for an arbitrary \( e: \text{Elt.}X \) and an arbitrary \( l: \text{List} \).

• (i) is called the base case, and (ii) the induction case.

• \( p(l) \) is called the induction hypothesis.

---

Formal Proof of Associativity of \(_@_\)

**Theorem** \( (L_1 @ L_2) @ L_3 = L_1 @ (L_2 @ L_3) \) for all \( L_1, L_2, L_3: \text{List} \).

**Proof** By induction on \( L_1 \).

Let \( x, y, z \) be arbitrary natural numbers.

I. Base case
   - open THEOREM-@
   - check
     - red \( \text{thl(nil,12,13)} \).
   - close

II. Induction case
   - open THEOREM-@
   - arbitrary values
     - op \( e : \rightarrow \text{Elt.}X \).
   - check
     - red \( \text{thl(11,12,13) implies thl(e \mid 11,12,13)} \).
   - close

QED

CafeOBJ returns true for both cases.
Revision of LIST based on Proofs

- It is also possible to prove that \texttt{nil} is a right identity of \_@\_ likewise.
- LIST is revised to reflect the two properties as follows:

```plaintext
mod! LIST (M :: TRIV) {
    [List]
    op nil : -> List
    op |_| : Elt.M List -> List
    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 .
}
```

A Simple Calculator

- It deals with the following expressions:
  - \texttt{n} (natural numbers),
  - \texttt{e}_1 ++ \texttt{e}_2 (addition),
  - \texttt{e}_1 -- \texttt{e}_2 (subtraction),
  - \texttt{e}_1 ** \texttt{e}_2 (multiplication), and
  - \texttt{e}_1 // \texttt{e}_2 (division),
  
where \texttt{e}_1 and \texttt{e}_2 are expressions.

- Two versions of a calculator are made:
  - one that interprets an expression (\textit{interpreter}), and
  - the other that generates a sequence of instructions from an expression and executes it (\textit{compiler + virtual machine}).
Expressions

• They are specified in module EXP.

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

    • l-assoc specifies that $e_1 + e_2 + e_3$ is parse as $(e_1 + e_2) + e_3$.

Interpreter (1)

• It is specified in module INTERPRETER.

    mod! INTERPRETER {
        pr(EXP)
        op interpret : Exp -> ErrorNat
        var N : Nat
        vars E E1 E2 : Exp
        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) .
    }
Interpreter (2)

- **Examples**
  - \(\text{interpret}(s(s(0)) ++ s(s(0)) ** s(s(0)))\)
  - \(s(s(s(s(s(0)))))\)
  - \(\text{interpret}(0 // 0)\)
    - \(\text{errorNat}\)
  - \(\text{interpret}(0 // 0 ++ 0)\)
    - \(\text{errorNat}\)

Instructions

- **Instructions (or commands) are specified in module \texttt{COMMAND}**.

  ```plaintext
  mod! \texttt{COMMAND} { 
  p\{\text{PNAT}\} 
  [\text{Command}] 
  op push : Nat -> Command 
  op multiply : -> Command 
  op divide : -> Command 
  op add : -> Command 
  op minus : -> Command 
  }
  ``

- **Sequences of instructions are specified in module \texttt{CLIST}**.

  ```plaintext
  mod! \texttt{LIST} { 
  p\{\text{LIST(\texttt{TRIV2COMMAND}) * (sort List -> CList)}\} 
  }
  ```
Virtual Machine (1)

- It is specified in module VM.
- A stack is used to execute a sequence of instructions.
- A list of natural numbers and errorNat is used as a stack.

\[
\text{LIST(TRIV2PNAT)}
\]
* {sort List -> Stack, op nil -> empstk}

- We have two operators:
  - \(\text{op \vm : CList -> ErrorNat}\)
  - \(\text{op \exec : CList Stack -> ErrorNat}\)
- Operator \(\vm\) is defined as follows:
  \[
  \text{eq \vm(CL) = \exec(CL, empstk)}
  \]

Virtual Machine (2)

- Operator \(\exec\) is defined as follows:
  \[
  \text{eq \exec(nil, N | empstk) = N} .
  \text{eq \exec(push(N) | CL, Stk) = \exec(CL, N | Stk)} .
  \text{eq \exec(inst | CL, N2 | N1 | Stk) = \exec(CL, natExp | Stk)} .
  \]
  \[
  \text{where inst is add, minus, multiply and natExp is N1 + N2,}
  \text{sd(N1,N2), N1 * N2 and N1 quo N2, respectively.}
  \]
- For any other pattern of a sequence \(cl\) of instructions and a stack \(stk\) (i.e. \(\exec(cl, stk)\)), \(\exec\) returns errorNat such as
  \[
  \text{eq \exec(add | CL, empstk) = errorNat} .
  \text{eq \exec(CL, errorNat | Stk) = errorNat} .
  \]
Compiler (1)

- It is specified in module COMPILER.
- We have one operator.
  \[
  \text{op \ compile : Exp \to CList}
  \]
- The operator is defined as follows:
  \[
  \begin{align*}
  \text{eq \ compile}(N) &= \text{push}(N) \mid \text{nil} \\
  \text{eq \ compile}(E_1 \ bop \ E_2) &= \text{compile}(E_1) @ \text{compile}(E_2) @ (\text{inst} \mid \text{nil})
  \end{align*}
  \]
  where \( bop \) is ++, --, ** and //, and \( \text{inst} \) is add, minus, multiply and divide, respectively.

Compiler (2)

- Examples
  - \( \text{compile}(s(s(0)) ++ s(s(0)) \,**\, s(s(0))) \)
    \[
    \text{push}(s(s(0))) \mid \text{push}(s(s(0))) \mid \text{push}(s(s(0))) \mid \text{multiply} \mid \text{add} \mid \text{nil}
    \]
  - \( \text{compile}(0 // 0) \)
    \[
    \text{push}(0) \mid \text{push}(0) \mid \text{divide} \mid \text{nil}
    \]
  - \( \text{compile}(0 // 0 ++ 0) \)
    \[
    \text{push}(0) \mid \text{push}(0) \mid \text{divide} \mid \text{push}(0) \mid \text{add} \mid \text{nil}
    \]
  - \( \text{vm}(\text{compile}(s(s(0)) ++ s(s(0)) \,**\, s(s(0)))) \)
    \[
    s(s(s(s(s(0)))))
    \]
  - \( \text{vm}(\text{compile}(0 // 0)) \)
    \[
    \text{errorNat}
    \]
  - \( \text{vm}(\text{compile}(0 // 0 ++ 0)) \)
    \[
    \text{errorNat}
    \]
Correctness of Compiler

• We define the correctness of the compiler as follows:
  “For all expressions $exp$, if the interpreter returns a natural number for $exp$, then the virtual machine returns exactly the same natural number for the sequence of instructions generated by the compiler for $exp$.”

• This is formalized as follows:
  $\forall e: \text{Exp}. \forall n: \text{Nat}. (\text{interpret}(e) = n \Rightarrow \text{vm}(\text{compile}(e)) = n)$

Proof by Induction on Expressions

• For a function (predicate) $p : \text{Exp} \rightarrow \text{Bool}$, the following two formulas are equivalent:
  1. $p(E)$ for all $E: \text{Exp}$
  2. $p(N), p(E_1)$ and $p(E_2)$ implies $p(E_1 + E_2), \ldots, p(E_2)$ implies $p(E_1 // E_2)$ for all $N: \text{Nat}$ and $E_1, E_2: \text{Exp}$.

• Therefore, to prove (1), it suffices to show
  (i) Base case: $p(n)$ for an arbitrary $n: \text{Nat}$.
  (ii) Induction case:
      1. $p(e_1 + e_2)$ assuming $p(e_1), p(e_2)$ for arbitrary $e_1, e_1: \text{Exp}$.
      ...
      4. $p(e_1 // e_2)$ assuming $p(e_1), p(e_2)$ for arbitrary $e_1, e_1: \text{Exp}$.

• (i) is called the base case, and (ii) the induction case.
• $p(e_1)$ and $p(e_2)$ are called the induction hypotheses.
Verification of Compiler (1)

• The following are declared in module THEOREM-COMPILER:
  
  ops \texttt{e1 \ e2} : \rightarrow \text{Exp} 
  
  ops \texttt{n \ m} : \rightarrow \text{Nat} 
  
  op \texttt{th1} : \text{Exp} \text{ Nat} \rightarrow \text{Bool} 
  
  eq \texttt{th1}(E,N) = (\text{interpret}(E) = N \implies \text{vm}(\text{compile}(E)) = N) . 
  
  • We prove \texttt{th1}(E,N) (for all expressions \texttt{E} and all natural numbers \texttt{N}) by induction on \texttt{E}.

Verification of Compiler (2)

• The proof passage of the base case is as follows:
  
  open THEOREM-COMPILER
  
  -- check
  
  red \texttt{th1}(m,n) .
  
  close
  
  CafeOBJ returns true for the proof passage.

• The initial proof passage of the induction case for \texttt{\_++\_} is as follows:
  
  open THEOREM-COMPILER
  
  -- check
  
  red \texttt{th1}(e1 ++ e2,n) .
  
  close
  
  CafeOBJ returns neither true nor false.
  
  Since \text{interpret(e1)} and \text{interpret(e2)} appears in the result, and they return errorNat or a natural number, then we split the case into four sub-cases based on the values returns by \text{interpret(e1)} and \text{interpret(e2)}. 

Verification of Compiler (3)

- The four sub-cases are as follows:

<table>
<thead>
<tr>
<th>Case</th>
<th>Interpret(e1)</th>
<th>Interpret(e2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>errorNat</td>
<td>errorNat</td>
</tr>
<tr>
<td>Case 2</td>
<td>a natural number k1</td>
<td>errorNat</td>
</tr>
<tr>
<td>Case 3</td>
<td>errorNat</td>
<td>a natural number k2</td>
</tr>
<tr>
<td>Case 4</td>
<td>a natural number k1</td>
<td>a natural number k2</td>
</tr>
</tbody>
</table>

- CafeOBJ returns true for the first three cases, but neither true nor false for the last one.
- The proposition $k_1 + k_2 = n$ appers in the result for case 4 and then the case is split into two sub-cases.

| Case 4-1 | false |
| Case 4-2 | true |

Verification of Compiler (4)

- CafeOBJ returns true for case 4-1, but the following for case 4-2.
  \[
  \text{exec}((\text{compile}(e1) @ (\text{compile}(e2) @ (\text{add | nil}))), \text{empstk}) = n
  \]
  This proposition is referred as (1).
- We can conjecture the following:
  - (2) implies (1)
  - (3) implies (2)
  where (2) and (3) are as follows:
  \[
  \text{exec}((\text{compile}(e2) @ (\text{add | nil})), \text{interpret}(e1) | \text{empstk}) = n
  \]
  \[
  \text{exec}(\text{add | nil}, \text{interpret}(e2) | \text{interpret}(e1) | \text{empstk}) = n
  \]
- (3) is equivalent to $k_1 + k_2 = n$ when $\text{interpreter}(e1)$ and $\text{interpreter}(e2)$ return natural numbers $k_1$ and $k_2$. 
Verification of Compiler (5)

• The conjecture lets us come up with the lemma:
  \[
  \text{op th2 : Exp CList Stack Nat \to Bool}
  \]
  \[
  \text{eq th2(E,L,S,N)}
  = \ (\text{exec(L,interpret(E) | S) = N)}
  \]
  \[
  \text{implies exec(compile(E) @ L,S) = N)}
  \]
  
• The conjecture says that
  – If (2) is assumed, then an instance of the lemma is the exactly the same as (1), discharging the case.
  – If the negation of (2) is assumed, then an instance of the lemmas becomes false because of \( k_1 + k_2 = n \), discharging the case.
  
• Then, case 4-2 is split into two sub-cases based on (2).

| Case 4-2-1 | True |
| Case 4-2-2 | False |

(2) is as follows:

\[
\text{exec(compile(e2) @ (add | nil), interpret(e1) | empstk) = n }
\]

Verification of Compiler (6)

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

```
open THEOREM-COMPILER
-- arbitrary values
ops k1 k2 : \to Nat .
-- assumptions
eq interpret(e1) = k1 .
\]
```

(2)
```
eq interpret(e2) = k2 .
eq k1 + k2 = n .
eq \text{exec((compile(e2) @ (add | nil)),(k1 | empstk))} = n .
```

(2) Implies (1)
```
-- check
red \(\text{th2(e1,compile(e2) @ (add | nil),empstk,n)}\)
implies \(\text{th1(e1 ++ e2,n)}\) .
```

(1)
```
close
```

CafeOBJ returns true for this proof passage.
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 . (3)
  eq interpret(e2) = k2 . (2)
  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

CafeOBJ returns true for this proof passage.

Verification of Compiler (8)

• The remaining three sub-cases of the induction case can be proved likewise.
• The lemma can be proved without any other lemmas.
Exercise

• Complete the proof.