Introduction to Specification & Verification in CafeOBJ

1. Some Basics of CafeOBJ

Kazuhiro Ogata
(JAIST)

Gist

- Some basic concepts such as modules and sorts on CafeOBJ are given.
- Natural numbers plus some operators are specified in CafeOBJ.
- It is proved that natural numbers enjoy some properties by writing proof scores in CafeOBJ.
Roadmap

• Some basic concepts on CafeOBJ
  – algebras, modules, sorts, operators, variables
  – terms, mixfix operators, equations, reduction
  – operator attributes, conditional equations
  – module importation

• Proof score approach to verification
  – proof scores
  – proofs on some properties on natural numbers

CafeOBJ

• An algebraic specification language/system.
• Mainly developed at JAIST; the project has been led by Kokichi Futatsugi.
• A direct successor of OBJ3; one of the main designer is Joseph A. Goguen.
• CafeOBJ specification can be executed; this executability makes it possible to verify that some properties hold for specifications (or specifications enjoy some properties).
Algebras

• Sets plus functions (or operators) over the sets.
  – The set \{0, 1, \ldots\} of natural numbers and some functions (e.g. addition (+) and multiplication (\ast)) are an example of algebras.
  – The set \{true, false\} of Boolean values and some functions (e.g. conjunction (\land) and disjunction (\lor)) are another example of algebras.
• A computer system such as computer software consists of data values and things that deal with the values, and then can be regarded as an algebra.
• Therefore, a computer system can be naturally specified as an algebra in an algebraic specification language.

Overview of CafeOBJ Specifications (1)

• A CafeOBJ specification consists of modules.
• An example of modules is as follows:

```plaintext
mod! PNAT {
  Sorts
  [Nat]
  op \_ 0 : \rightarrow Nat
  op \_ s : Nat \rightarrow Nat
  op \_ + : Nat Nat \rightarrow Nat

  Operators
  vars \_ X \_ Y : Nat
  eq \_ 0 + Y = Y .
  eq \_ s(X) + Y = s(X + Y) .
}
```
Modules

- *Modules* are basic units of CafeOBJ specifications.
- They are declared as follows:
  \[
  \text{mod! } \text{ModuleName} \{ \ldots \}
  \]
- In the body (marked as ...) of the module, the following are declared:
  - *Sorts*
  - *Operators*
  - *Variables*
  - *Equations*

Sorts

- Names given to sets (corresponding to types in other programming languages).
  - \text{Nat} is the sort given to the set \{0, 1, \ldots\} of natural numbers.
  - \text{Bool} is the sort given to the set \{true, false\} of Boolean values.
- Sorts are declared by enclosing [ and ].
  - [Nat]
  - [Bool]
Operators

- **Operators** are declared as follows:
  \[ \text{op } f : S_1 \ldots S_n \rightarrow S \]
- The sequence \( S_1 \ldots S_n \) of sorts is called the *arity* of \( f \);
  the sort \( S \) is called the *sort* (or *coarity*) of \( f \), and the pair
  of the arity and sort is called the *rank* of \( f \).
- When the arity is empty (i.e. \( n = 0 \)), the operator is
called a *constant*.
- Multiple functions whose ranks are the same can be
declared simultaneously.
  \[ \text{ops } f_1 \ldots f_m : S_1 \ldots S_n \rightarrow S \]

Variables

- Variables are declared as follows:
  \[ \text{var } X : S \]
  \( X \) is a variable whose sort is \( S \).
- Multiple variables whose sorts are the same
can be declared simultaneously.
  \[ \text{vars } X_1 \ldots X_m : S \]
Terms

- *Terms* are inductively defined as follows:
  - A variable $X$ of sort $S$ is a term of sort $S$.
  - For a function $f$ whose rank is $\mathcal{S}_1 ... \mathcal{S}_n \rightarrow S$ and terms $t_1, ..., t_n$ of sorts $\mathcal{S}_1, ..., \mathcal{S}_n$, $f(t_1, ..., t_n)$ is a term of sort $S$.

- Examples from module PNAT:
  - $X$ and $Y$ are terms of sort $\text{Nat}$.
  - $0$ is a term of $\text{Nat}$.
  - $s(0), s(s(0)), ...$ are terms of $\text{Nat}$.

Mixfix Operators

- The positions where arguments of operators are put can be freely designed and then context-free grammars of terms can be defined.

- Examples
  - infix operators
    - $\text{op \_+\_ : Nat Nat \rightarrow Nat}$
  - prefix operators
    - $\text{op \~\_ : Bool \rightarrow Bool}$
  - postfix operators
    - $\text{op \_! : Nat \rightarrow Nat}$
  - more general mixfix operators
    - $\text{op if\_then\_else\_fi : Bool Nat Nat \rightarrow Nat}$

- $\text{if ~ B then X + Y else Z \_! fi}$ is a term of $\text{Nat}$, where $B$ is a variable of $\text{Bool}$, and $X, Y, Z$ are variables of $\text{Nat}$.

- An underscore _ indicates a place where an argument is put, and the number of underscores must be the same as the number of arguments.
Equations

• *Equations* are used to define functions (operators) and properties of operators.
• Equations are declared as follows:
  \[
  \text{eq } \ term_1 = \ term_2 .
  \]
  – \( \term_1 \) and \( \term_2 \) are terms of a same sort.
  – \( \term_1 \) is not a single variable.
  – All variables in \( \term_2 \) must appear in \( \term_1 \).
• Example: the following two equations define the addition operator of natural numbers:
  \[
  \text{eq } 0 + Y = Y .
  \]
  \[
  \text{eq } s(X) + Y = s(X + Y) .
  \]

Reduction (1)

• Equations are regarded as left-to-right rewrite rules to *reduce (or simplify or compute)* a given term.
• *One-step rewrite* is to replace an instance (called a *redex*) of the LHS of an equation in a given term with the corresponding instance (called a *contract*) of the RHS.

\[
\begin{align*}
\text{s (s(0) + s(s(0)))} & \rightarrow \text{s (s(0 + s(s(0)))}) \\
\text{A redex (an instance of the LHS of (+2))} & \quad \text{A contract (the corresponding instance of the RHS)}
\end{align*}
\]

\[
\begin{align*}
\text{eq } 0 + Y = Y . & \quad -- (+1) \\
\text{eq } s(X) + Y = s(X + Y) . & \quad -- (+2)
\end{align*}
\]
Reduction (2)

- Reduction is done by applying one-step rewrite to a given term while there are redexes in the term and returns a term (called a canonical form of the given term) that has no redexes.
- A procedure to reduce a given term is like
  \[ t := \text{a given term}; \]
  \[ \text{while } 3 \text{a redex } r \text{ in } t \text{ do} \]
  \[ t := \text{a term obtained by replacing } r \text{ with the contract}; \]
  \[ \text{od;} \]
  \[ \text{return } t; \]
- Command "red in } M : t ." reduces a given term \( t \) wrt module \( M \) (or the set of equations).
- "red in PNAT : s(s(0)) + s(s(0)) ." returns the following:
  \( (s(s(s(s(0))))):\text{Nat} \)

Reduction (3)

- Command "set trace on" has command "red in } M : t ." show each one-step rewrite performed in the reduction such as

  1> [1] rule: eq (s(X:Nat) + Y:Nat) = s((X + Y))
  \{ X:Nat |\rightarrow s(0), Y:Nat |\rightarrow s(s(0)) \} 
  1< [1] (s(s(0)) + s(s(0))) --\rightarrow s((s(0) + s(s(0))))

  1> [2] rule: eq (s(X:Nat) + Y:Nat) = s((X + Y))
  \{ X:Nat |\rightarrow 0, Y:Nat |\rightarrow s(s(0)) \} 
  1< [2] (s(0) + s(s(0))) --\rightarrow s((0 + s(s(0))))

  1> [3] rule: eq (0 + Y:Nat) = Y
  \{ Y:Nat |\rightarrow s(s(0)) \} 
  1< [3] (0 + s(s(0))) --\rightarrow s(s(0))

- Command "set trace off" has the reduction command hide such information.
Operator Attributes

- Operators can be given attributes, among which are prec: \( n \) and comm.
- \( \text{prec: } n \) specifies the precedence of the operator. The less the number \( n \), the stronger the precedence.
  
  \[
  \text{op } _+_: \text{Nat Nat } \rightarrow \text{Nat } \{\text{prec: 30}\}
  \text{op } _*_: \text{Nat Nat } \rightarrow \text{Nat } \{\text{prec: 29}\}
  \]
  
  \( x + y * z \) is parsed as \( x + (y * z) \).
- \( \text{comm} \) specifies that the operator is commutative.
  
  \[
  \text{op } _=: \text{Nat Nat } \rightarrow \text{Bool } \{\text{comm}\}
  \]
  
  \( x = y \) equals \( y = x \).

Built-in Module BOOL

- Sort \( \text{Bool} \) denoting the truth values is declared.
- Constants \( \text{true} \) and \( \text{false} \) are declared.
- Among other operators declared are:
  
  - \( \text{and} \): conjunction
  - \( \text{or} \): disjunction
  - \( \text{not} \): negation
  - \( \text{implies} \): implication
  - \( \text{xor} \): exclusive disjunction
Conditional Equations

• Equations can be equipped with conditions.
• Equations with conditions are called conditional equations.
• Conditional equations are declared as follows:
  \[ \text{ceq } term_1 = term_2 \text{ if } \text{cond}. \]
  – \( \text{cond} \) is a term of sort \( \text{Bool} \).
  – All variables in \( \text{cond} \) must appear in \( \text{term}_1 \).
  – The equation can be used only if \( \text{cond} \) reduces to \text{true}.
• Example (Factorial function):
  \[ \text{ceq } X! = s(0) \text{ if } X = 0. \]
  \[ \text{eq } s(X)! = s(X) \ast (X!). \]
  The 1st equation can be replace with the following:
  \[ \text{eq } 0! = s(0). \]

Importation of Modules

• Modules can be imported by other modules in which imported modules can be used.
• Module \( M \) is imported as follows:
  \[ \text{pr}(M). \]
• Almost all modules automatically imports built-in module \( \text{BOOL} \).
• Example

  mod! FACT {
    \[ \text{pr}(\text{PNAT}). \]
    \[ \text{op }_!: \text{Nat} \to \text{Nat}. \]
    \[ \text{var } X : \text{Nat}. \]
    \[ \text{ceq } X! = s(0) \text{ if } X = 0. \]
    \[ \text{eq } s(X)! = s(X) \ast (X!). \]
  }

In module FACT, some of what are declared in module \text{PNAT} are used.
Commands open & close

- Command open makes a temporary module in which a given module is imported, and command close destroys such a temporary module.
- In open-close sections, what are declared in usual modules can be declared, and command red can also be used.
- Example
  
  open FACT
  red 0 ! .
  red s(s(s(0))) ! .
  close

Some Properties on _+_ and _*_*

- We know that addition and multiplication of natural numbers are associative and commutative.
- Is it possible to prove that _+_ and _*_* are also associative and commutative based on module PNAT?
- Let us try to prove them.
Proof by Induction on Natural Numbers

- For an operator (predicate) \( p : \text{Nat} \rightarrow \text{Bool} \), the following two formulas are equivalent:
  1. \( p(N) \) for all \( N: \text{Nat} \)
  2. \( p(0) \) and \( p(N) \) implies \( p(s(N)) \) for all \( N: \text{Nat} \)

- Therefore, to prove (1), it suffices to show
  1. \( p(0) \)
  2. \( p(s(n)) \) assuming \( p(n) \) for an arbitrary \( n: \text{Nat} \)

- (i) is called the base case, and (ii) the induction case.
- \( p(n) \) is called the induction hypothesis.

Proof of Associativity of \( _+\_ \)

**Theorem** \( (X + Y) + Z = X + (Y + Z) \) for all \( X, Y, Z: \text{Nat} \).

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

Let \( X, Y, Z \) be arbitrary natural numbers.

I. **Base case**
   - All we have to do is to show \( (0 + y) + z = 0 + (y + z) \).
     - \( \text{LHS} \rightarrow y + z \) (by \(+1\))
     - \( \text{RHS} \rightarrow y + z \) (by \(+1\))

II. **Induction case**
   - All we have to do is to show \( s(x) + y) + z = s(x) + (y + z) \)
     - assuming the induction hypothesis \( (x + y) + Z = x + (Y + Z) \)
     - for all \( Y, Z: \text{Nat} \).
     - \( \text{LHS} \rightarrow s(x + y) + z \) (by \(+2\)) \( \rightarrow s((x + y) + z) \) (by \(+2\))
     - \( \text{RHS} \rightarrow s((x + (y + z)) \) (by \(+1\)) \( \rightarrow s((x + y) + z) \) (by \(+1\))

QED
Formal Proof of Associativity of \( _+_{ } \) (1)

- Module \textsc{Theorem-Pnat} is first declared as follows:

```markdown
mod \textsc{Theorem-Pnat} {
  pr(PNAT)  
  -- arbitrary values
  ops (x y z): -> Nat .
  -- Names of Theorems
  op th1 : Nat Nat Nat -> Bool
  -- CafeOBJ variables
  vars X Y Z : Nat
  -- Theorems
  eq th1(X, Y, Z)
  = ((X + Y) + Z = X + (Y + Z)) .
}
```

Formal Proof of Associativity of \( _+_{ } \) (2)

\textbf{Theorem} \( (X + Y) + Z = X + (Y + Z) \) for all \( X, Y, Z: \text{Nat} \).

\textbf{Proof} By induction on \( X \).
Let \( x, y, z \) be arbitrary natural numbers.

I. Base case
   - open \textsc{Theorem-Pnat}  
   -- check
   ```markdown
   red th1(0, y, z) .
   ```
   close

II. Induction case
   - open \textsc{Theorem-Pnat}  
   -- check
   ```markdown
   red th1(x, y, z) implies th1(s(x), y, z) .
   ```
   close

\textsc{QED}

An instance of the induction hypothesis \( th1(x, y, z) \) for all \( Y, Z: \text{Nat} \) obtained by replacing \( Y \) and \( Z \) with \( y \) and \( z \).
Proof Scores

• Proofs (or proof plans) written in CafeOBJ are called \textit{proof scores}.
• Proof scores consist of fragments enclosed with open and close.
• Fragments enclosed with open and close, which constitute proof scores, are called \textit{proof passages}.

A proof score of \texttt{th1(X,Y,Z)} for all \texttt{X,Y,Z:Nat} consisting of two proof passages:

\begin{verbatim}
open THEOREM-PNAT -- check
red th1(0,y,z) .
Close
open THEOREM-PNAT -- check
red th1(x,y,z)
  implies th1(s(x),y,z) .
close
\end{verbatim}

Formal Proof of Commutativity of \texttt{\_ + \_} (1)

• In module \texttt{THEOREM-PNAT}, the following are declared:
  \begin{verbatim}
op th2 : Nat Nat -> Bool
  eq th2(X,Y) = (X + Y = Y + X) .
\end{verbatim}
• What to prove is \texttt{th2(X,Y)} for all \texttt{X,Y:Nat}.
• The proof is done by induction on \texttt{X}.
• The initial proof passage of Base case is as follows:
  \begin{verbatim}
open THEOREM-PNAT -- check
  red th2(0,y) .
close
CafeOBJ returns \texttt{y = (y + 0)}, which must be true.
\end{verbatim}
• But, CafeOBJ does not have enough information to conclude this.
• We then conjecture the lemma:
  \begin{verbatim}
eq th5(X) = (X + 0 = X) .
\end{verbatim}
Formal Proof of Commutativity of \( _+_ \) (2)

- The lemma \( \text{th5}(X) \) for all \( X: \text{Nat} \) is first proved by induction on \( X \).
  
  I. Base case
  
  open THEOREM-PNAT
  -- check
  red \( \text{th5}(0) \) .
  close
  
  II. Induction case
  
  open THEOREM-PNAT
  -- check
  red \( \text{th5}(x) \) implies \( \text{th5}(s(x)) \) .
  close
  
- CafeOBJ returns true for both proof passages.
- Hence, the proof is successfully done.

Formal Proof of Commutativity of \( _+_ \) (3)

- Then, the proof passage of Base case becomes as follows:
  
  open THEOREM-PNAT
  -- check
  red \( \text{th5}(y) \) implies \( \text{th2}(0,y) \) .
  close  
  CafeOBJ returns true.
  
- The initial proof passage of Induction case is as follows:
  
  open THEOREM-PNAT
  -- check
  red \( \text{th2}(x,y) \) implies \( \text{th2}(s(x),y) \) .
  close  
  CafeOBJ returns the following:
  
  \( (x + y = y + x \text{ and } s(x + y) = y + s(x)) \)
  \( \text{xor} \ (x + y = y + x \text{ xor true}) \)
  
  Then, the proposition \( x + y = y + x \) is used to split the case into two sub-cases: (1) it is false, and (2) it is true.
Formal Proof of Commutativity of \(+\) (4)

- The proof passage of sub-case (1) where the proposition is false is as follows:
  open THEOREM-PNAT
  -- assumptions
  eq (x + y = y + x) = false .
  -- check
  red th2(x,y) implies th2(s(x),y) .
  close
  CafeOBJ returns true.
- Hence, this sub-case is discharged.

Formal Proof of Commutativity of \(+\) (5)

- The proof passage of sub-case (2) where the proposition is true is as follows:
  open THEOREM-PNAT
  -- assumptions
  eq x + y = y + x .
  -- check
  red th2(x,y) implies th2(s(x),y) .
  close
  CafeOBJ returns s(y + x) = y + s(x), which must be true.
- But, CafeOBJ does not have enough information to conclude this.
- We then conjecture the lemma:
  eq th6(X,Y) = (X + s(Y) = s(X + Y)) .
Formal Proof of Commutativity of _+_ (6)

- The lemma \( \text{th6}(X,Y) \) for all \( X,Y: \text{Nat} \) is proved by induction on \( X \).
  I. Base case
  open \( \text{THEOREM-PNAT} \)
  -- check
  red \( \text{th6}(0,y) \).
  close
  II. Induction case
  open \( \text{THEOREM-PNAT} \)
  -- check
  red \( \text{th6}(x,y) \) implies \( \text{th6}(s(x),y) \).
  close
- \( \text{CafeOBJ} \) returns \text{true} for both proof passages.
- Hence, the proof is successfully done.

Formal Proof of Commutativity of _+_ (7)

- Then, the proof passage of sub-case (2) becomes as follows:
  open \( \text{THEOREM-PNAT} \)
  -- assumptions
  eq \( x + y = y + x \).
  -- check
  red (\( \text{th2}(x,y) \) and \( \text{th6}(y,x) \))
  implies \( \text{th2}(s(x),y) \).
  close
- \( \text{CafeOBJ} \) returns \text{true}.
- We have proved that _+_ is associative and commutative.
- It is also possible to prove that _*_ is associative and commutative likewise.
Exercise

• Prove that \( _*_* \) is associative and commutative by writing proof scores.