Basics of CafeOBJ (2)
- verification and modeling -

CafeOBJ Team of JAIST

Topics

- Basics of verification with CafeOBJ
  - Reduction, induction, arbitrary constants, variables

- An example of modeling in CafeOBJ
  - Stack with error handling
How to do verification with CafeOBJ specifications

- The basic mechanism of CafeOBJ verification is equational reasoning. Equational reasoning is to deduce an equation (a candidate of a theorem) from a given set of equations (axioms or specifications).
- The CafeOBJ system supports an automatic equational reasoning based on rewriting (or TRS: Term Rewriting System).
- “reduce” or “red” (reduction) command to do equational reasoning is provided by CafeOBJ System.
Conditions for an equation to be a rewriting rule

For an equation to be used as a rewriting rule for doing reductions, the following conditions must be satisfied.

1. LHS is not a variable.
   an example violating this condition:
   \[ \text{eq } N: \text{Nat} = +(N: \text{Nat}, 0) . \]

2. All variables in RHS are in LHS.
   an example violating this condition:
   \[ \text{eq } 0 = *(N: \text{Nat}, 0) . \]

Reduction command:
Equational reasoning by rewritings

There are two ways to do equational reasoning in CafeOBJ by rewritings: \text{red } \langle \text{term} \rangle. \text{ and } \text{red } \langle \text{term} \rangle = \langle \text{term} \rangle.

\begin{verbatim}
NAT+> red +(0, s(0)) .
-- reduce in NAT+ : +(0, s(0))
s(0) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)
\end{verbatim}

This means that the input term is equivalent to the output term.

\begin{verbatim}
NAT+> open (NAT+ + EQL) -- for using equality (_=_)
NAT+ + EQL> red +(0, s(0)) = +(s(0), 0) .
-- reduce in NAT+ : +(0, s(0)) = +(s(0), 0)
true : Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 5 matches)
\end{verbatim}

This means that the one side is equivalent to the other side.
What can be done with \texttt{red (reduction)} command?

A reduction command of CafeOBJ:

\begin{verbatim}
MODULE> red inputTerm .
\end{verbatim}

returns a most simplified term of the given term \textit{inputTerm} by using all equations of the module \texttt{MODULE} as rewriting rules from LHS to RHS. For any context, \texttt{any-module} \texttt{> red in MODULE : red inputTerm .}
returns the same result.

Let us fix a context M (a module M in CafeOBJ), and let \((t1 =^{*M}> t2)\) denote that \(t1\) is reduced to \(t2\) in the context. That is, \((\text{red in M : t1 .})\) returns \(t2\). Let \((t1 =^M t2)\) denote that \(t1\) is equal to \(t2\) in the context \(M\). It is important to notice:

\((t1 =^{*M}> t2)\) implies \((t1 =^M t2)\)

but

\((t1 =^M t2)\) does not implies \((t1 =^{*M}> t2)\)

---

Two equality predicates \(=_=\) and \(===\)

Assume that \((t1 =^{*} t1')\) and \((t2 =^{*} t2')\) in any context then

if \((t1'\ \text{and}\ t2'\ \text{are the same term})\)
then \((\text{red t1 = t2 .})\) returns true

and

\((\text{red t1 == t2 .})\) returns true

if \((t1'\ \text{and}\ t2'\ \text{are different terms})\)
then \((\text{red t1 = t2 .})\) returns \((t1' = t2')\)

but

\((\text{red t1 == t2 .})\) returns false
Soundness of \_==\_ and \_=\_

- The result of “red \_\text{term1}\_ == \_\text{term2}\_ .” is sound but not complete, that is:
  - If it returns true, then the two terms \_\text{term1}\_ and \_\text{term2}\_ is proved to be equal.
  - But if it returns false, then the two terms may equal or not equal.

- The reduction of Boolean term involving \_==\_ may return true even if it is not true w.r.t. the set of axioms (or the specification). That is, \_==\_ may not be sound.

- If the reduction of Boolean term involving only \_=\_ returns true, then it is true w.r.t. the set of axioms (or the specification).

Proof scores

- A fragment proof score begins at “open” command which opens a module, and ends with “close” command.
- While a module is opened (between open and close), we can declare operations and equations for doing verification.

```plaintext
\text{NAT+}\rightarrow\text{open (NAT+ + EQL)}
-- opening module NAT+.. done.
%NAT+ + EQL> op n : -> Nat .
%NAT+ + EQL> eq n = 0 .
%NAT+ + EQL> red +(n, n) = 0 .
* -- reduce in %NAT+ + EQL : +(n,n) = 0
true : Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 4 matches)
%NAT+ + EQL> close
NAT+>
```
**Arbitrary element**

- After opening a module, a declared constant operation
  \[
  \text{op e : } \rightarrow S .
  \]
  stands for an arbitrary element of the sort \( S \) whose scope is from its declaration to the end of a proof score (i.e. \texttt{close}).

\[
\text{NAT+> open (NAT+ + EQL)}
\]

-- opening module NAT+.. done.

%NAT+ + EQL> op n : -> Nat .

%NAT+ + EQL> red +(0, n) = n .

-- reduce in %NAT+ : +(0,n) = n

true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)

%NAT+ + EQL> close

NAT+>

This is a proof score for the claim that \(+ (0, N) = N\) for any natural number \( N \). Since the reduction returns "true", it holds.

**Declaring assumptions**

- While a module is opening, a declared equation represents an assumption of the proof score.

\[
\text{NAT+> open (NAT+ + EQL)}
\]

-- opening module NAT+.. done.

%NAT+ + EQL> op n : -> Nat .

%NAT+ + EQL> eq +(n, 0) = n .

-- reduce in %NAT+ : +(n,0) = n

true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)

This is a proof for \("+(N, 0) = N implies +(s(N), 0) = s(N)" \) for any natural number \( N \) (it holds).
Constant v.s. variable

- Using a variable in an equation instead of a constant makes a drastic change of meaning of the proof score. Be careful!
  - The scope of a constant is to the end of a open-close session assuming that the declared constants are fresh.
  - The scope of a variable is inside of the equation.

\[
\text{open (NAT+ + EQL)}
\begin{align*}
\text{op n : } & \to \text{Nat} . \\
\text{eq } & +(n, 0) = n . \\
\text{red } & +(s(n), 0) = s(n) . \\
\text{close}
\end{align*}
\]

\[
\text{open (NAT+ + EQL)}
\begin{align*}
\text{var N : Nat} . \\
\text{eq } & +(N, 0) = N . \\
\text{red } & +(s(N), 0) = s(N) . \\
\text{close}
\end{align*}
\]

**Constant:** \(\forall N : \text{Nat. } \left[+(N, 0) = N \Rightarrow +(s(N), 0) = s(N) \right]\)

**Variable:** \(\forall N : \text{Nat. } \left[+(N, 0) = N \right] \Rightarrow \forall N : \text{Nat. } \left[+(s(N), 0) = s(N) \right]\)

---

Mathematical Induction over Natural Numbers

**Goal:** Prove that for any natural number \(n \in \{0, s\ 0, s\ s\ 0,\ldots\}\ P(n)\) is true

**Induction Scheme:**

\[
P(0) \quad \forall n \in \mathbb{N}. [P(n) \Rightarrow P(s\ n)]
\]

\[\forall n \in \mathbb{N}. P(n)\]

**Concrete Procedure:** (induction with respect to \(n\))

1. Prove \(P(0)\) is true
2. Assume that \(P(n)\) holds, and prove that \(P(s\ n)\) is true
Induction

- The following is a proof score of “∀n:Nat.+(n,0) = n”:

```
open (NAT+ + EQL)
red +(0, 0) = 0 .
op n : -> Nat .
eq +(n, 0) = n .
red +(s(n), 0) = s(n) .
close
```

```
-- opening module (NAT+ + EQL). done.
%NAT+ + EQL> -- reduce in %NAT+ + EQL : +(0,0) = 0
true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)

-- reduce in %NAT+ + EQL : +(s(n),0) = s(n)
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
%NAT+ + EQL> NAT+>
```

Complete proof score

```
--> This is a proof of +(N, 0) = N
open (NAT+ + EQL)
--> Base case
red +(0, 0) = 0 .
--> Induction step
op n : -> Nat .
eq +(n, 0) = n . -- I.H.
red +(s(n), 0) = s(n) .
close
```

```
NAT+> in nat+ps.mod
processing input : /.../proof.mod
--> This is a proof of +(N, 0) = N
-- opening module NAT+ + EQL. done.
--> Base case
-- reduce in %NAT+ + EQL : +(0,0) = 0
true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)

--> Induction step_4
-- reduce in %NAT+ + EQL : +(s(n),0) = s(n)
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
NAT+>
```
Trace commands

- When some reduction in a proof score does not return "true", the trace commands help us to detect the reason.

```plaintext
NAT+> set trace whole on
NAT+> red +(s(0), s(0)) .
-- reduce in NAT+ : +(s(0),s(0))
  [1]: +(s(0),s(0))
  ---> s(+(0,s(0)))
  [2]: s(+(0,s(0)))
  ---> s(s(0))
 s(s(0)) : NzNat
 (0.000 sec ...)
NAT+> set trace whole off

NAT+> set trace on
NAT+> red +(s(0), s(0)) .
-- reduce in NAT+ : +(s(0),s(0))
 1>[1] rule: eq +(s(M:Nat),N:Nat) = s(+(M,N))
    { M:Nat |-> 0, N:Nat |-> s(0) }
 1<[1] +(s(0),s(0)) --> s(s(0))
 1>[2] rule: eq +(0,N:Nat) = N
    { N:Nat |-> s(0) }
 1<[2] +(0,s(0)) --> s(0)
 s(s(0)) : NzNat
 (0.000 sec ...)
NAT+> set trace off
```

Modeling/Specifying in CafeOBJ

1. By understanding a problem to be modeled/specified, determine several sorts of objects (entities, data, agents, states) and operations (functions, actions, events) over them for describing the problem.

2. Define the meanings/functions of the operations by declaring equations over expressions composed of the operations.
Modeling STACK in CafeOBJ

Signature of STACK (1)

sorts of objects
- Element
- Stack

operations
- empty : returns empty stack without argument
- push : push an element to a stack and returns a new stack
- top : get the top element of a stack and returns the element
- pop : removes the top element of a stack and returns the stack

-- sorts
- [ Element Stack ]

-- operations or operators
- op empty : -> Stack
- op push : Element Stack -> Stack
- op pop_ : Stack -> Stack
- op top_ : Stack -> Element
Signature of STACK (2)

-- sorts
[ Element Stack ]
-- operations or operators
op empty : -> Stack
op push : Element Stack -> Stack
op pop_ : Stack -> Stack
op top_ : Stack -> Element

Terms/expressions generated by the signature

Element = \{ e1, e2, e3, \ldots \} 
U \{ top S \mid S \in Stack \}

Stack = \{ empty \} 
U \{ push(E,S) \mid E \in Element \land S \in Stack \}
U \{ pop S \mid S \in Stack \}

Examples:
top push(e1,empty) \in Element
pop push(e1,empty) \in Stack
top pop push(e1,empty) \in Element
pop pop push(e1,empty) \in Stack
top push(e2, pop pop push(e1,empty)) \in Element
Equations

-- operations or operators
op empty : -> Stack
op push : Element Stack -> Stack
op pop_ : Stack -> Stack
op top_ : Stack -> Element

-- equations
eq top push(E:Element,S:Stack) = E .
eq pop push(E:Element,S:Stack) = S .

Examples:
top push(e1,empty) = e1
pop push(e1,empty) = empty
top pop push(e1,empty) = top empty
pop pop push(e1,empty) = pop empty
top push(e2, pop pop push(e1,empty)) = e2
this is not the intended behavior

Revised signature of STACK (1)
Revised signature of STACK (2)

-- sorts and subsorts
[ Element, EmptyStack NonEmptyStack < Stack ]
-- operators
op empty : -> EmptyStack
op push : Element Stack -> NonEmptyStack
op pop_ : NonEmptyStack -> Stack
    -- only applicable to NonEmptyStack
op top_ : NonEmptyStack -> Element
    -- only applicable to NonEmptyStack

Terms generated by revised signature

Element = { e1, e2, e3, ... } 
    U { top S | S ∈ NonEmptyStack } 

EmptyStack = { empty } 
NonEmptyStack = { push(E,S) | E ∈ Element ∧ S ∈ Stack } 
Stack = { pop S | S ∈ NonEmptyStack } 
    U EmptyStack U NonEmptyStack

Examples:
top push(e1,empty) ∈ Element 
push(e1,empty) ∈ NonEmptyStack 
pop push(e1,empty) ∈ Stack 
top pop push(e1,empty) : not well formed!
pop pop push(e1,empty) : not well formed!
top push(e2, pop pop push(e1,empty)) : not well formed!
Revised Stack -- equations --

[ EmptyStack NonEmptyStack < Stack ]

op empty : -> EmptyStack {constr}

op push : Element Stack -> NonEmptyStack {constr}

op pop_ : NonEmptyStack -> Stack

op top_ : NonEmptyStack -> Element

-- equations

eq top push (E:Element, S:Stack) = E .

eq pop push (E:Element, S:Stack) = S .

Examples:

top push(e1,empty) = e1

pop push(e1,empty) = empty

top pop push(e1,empty) = (top empty : ?Element)

pop pop push(e1,empty) = (pop empty : ?Stack)

top push(e2, pop pop push(e1,empty)) =

(pop pop push(e1,empty) : ?Element)

Return value is of Error sort!

Revised specification of Stack in CafeOBJ

Specification of Stack

mod* ELEMENT { [Element] }

mod! STACK (X :: ELEMENT) {

[EmptyStack NonEmptyStack < Stack ]

op empty : -> EmptyStack {constr}

op push : Element Stack -> NonEmptyStack {constr}

op pop_ : NonEmptyStack -> Stack

-- only applicable to NonEmptyStack

op top_ : NonEmptyStack -> Element

-- only applicable to NonEmptyStack

eq top push (E:Element, S:Stack) = E .

eq pop push (E:Element, S:Stack) = S .

}