Verification with Induction

CafeOBJ Team of JAIST

Topics

- Explain how to prove properties by the induction techniques with CafeOBJ
  – Review: proof with an arbitrary element, etc
  – Several examples: Nat, List
  – Lemma discovery

Proof with an arbitrary element

- Review: Consider the following module

```
mod* PROOF-n {
  pr(NAT*)
  op n : -> Nat
}
```

```
PROOF-n + EQL> red s (0 + n) = 0 + s n .
(true):Bool
```

This is a proof of
"s(0 + **n**) = 0 + s **n**" **for any natural number n**

Soundness of the proof

- Consider the denotation of `PROOF-n`
  - includes a model of `NAT*` as it is (because of `pr`)
  - `NAT*` denotes Natural numbers algebra *N*
  - Constant `n` should be one of the elements of $N_{\mathtt{Nat}}$
  - For any natural number *x*, there exists a model M denoted by `PROOF-n` such that $M_n = x$

```
mod* PROOF-n {
  pr(NAT*)
  op n : -> Nat
}
```

```
PROOF-n> red s (0 + n) = 0 + s n .
(true):Bool
```

This is a proof of s(0 + n) = 0 + s n for any natural number n

Proof of Implication

- Consider the following module

```
mod* PROOF-i {
  pr(NAT*)
  ops x y : -> Nat
  eq x = y + y .
}
```

```
PROOF-i + EQL> red  x * s s 0 = (y * s s 0) + (y * s s 0) .
(true):Bool
```

This is a proof of
`x = s y` implies `x * s s 0 = (y * s s 0) + (y * s s 0)`

Soundness of the implication proof

- Consider the denotation of `PROOF-i`
  - `x` and `y` are elements of *N* and satisfy `x = s y` in a model
  - For any natural number *x* and *y* satisfying *"x = y + 1"*, there exists a model M denoted by `PROOF-i` such that $M_x = x$ and $M_y = y$

```
mod* PROOF-i {
  pr(NAT*)
  ops x y : -> Nat
  eq x = s y .
}
```

```
> red  x * s s 0
     = (y * s s 0) +
       (y * s s 0) .
(true):Bool
```

This is a proof of
`x = s y` implies `x * s s 0 = (y * s s 0) + (y * s s 0)`

Proof score (in the board sense)

- We can make nameless module for a proof by opening a module

```
PROOF-i + EQL> open NAT* + EQL
%NAT* + EQL> ops x y : -> Nat .
%NAT* + EQL> eq x = y + y .
%NAT* + EQL> red  x * s s 0 = (y * s s 0) + (y * s s 0) .
(true):Bool
%NAT* + EQL> close
PROOF-i + EQL>
```

proof score

```
open NAT* + EQL
  ops x y : -> Nat .
  eq x = y + y .
  red  x * s s 0 = ...
close
```

```
mod* PROOF-i {
  pr(NAT*)
  ops x y : -> Nat
  eq x = s y .}
```

```
PROOF-i + EQL> red  ...
```


Structural Induction

- Structural induction is a proof method for recursively-defined data structures (like terms)
  - To prove P(X) for all terms constructed by the set F of operators

1. [Induction Basis] Prove $P(c)$ for each constant c in F
2. [Induction Step] For each function f in F whose arity is n,
   - Assume $P(t_1)$, $P(t_2)$, ..., $P(t_n)$, and
   - Prove $P(f(t_1, t_2, ..., t_n))$

Example 1: Left-identity of +

- The following is a proof score of that 0 is a left-identity of +
  - P(N) = "0 + N = N"
  - Prove for all terms constructed by 0 and s_

```
open NAT+ + EQL
  red 0 + 0 = 0 .          Induction Basis
  op n : -> Nat .
  eq 0 + n = n   .         Induction Hypothesis
  red 0 + s n = s n .      Induction Step
close
```

CafeOBJ system returns `true` for both reductions for this proof score

```
mod! NAT+ {  ...
  eq N + 0 = N .
  eq M + s N = s(M + N) .
}
```


Trace of reduction

- You can see how I.H. is used in the proof

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

```
1>[1] rule: eq (M:Nat + (s N:Nat)) = (s (M + N))
    { M:Nat |-> 0, N:Nat |-> n }
[1]: 0 + (s n) = s n ---> s (0 + n) = s n

1>[2] rule: eq (0 + n) = n {}
[2]: s (0 + n) = s n ---> s n = s n

1>[3] rule: eq (CUX = CUX) = true
    { CUX |-> (s n) }
[3]: s n = s n ---> true
```

Wrong proof

- An arbitrary element in a induction proof score should not be declared as a variable

```
open NAT+ + EQL
  red 0 + 0 = 0 .
  var N : Nat .
  eq 0 + N = N   .
  red 0 + s N = s N .
close
```

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

$$\forall N : \mathrm{Nat}.\ [0 + N = N] \Rightarrow \forall N : \mathrm{Nat}.\ [0 + s\,N = s\,N]$$

$$\forall n : \mathrm{Nat}. \begin{bmatrix} 0 + n = n \\ \Rightarrow 0 + s\,n = s\,n \end{bmatrix}$$

```
1>[1] rule: eq (0 + N) = N { N |-> s N }
[1]: 0 + (s N) = s N ---> s N = s N
[2]: s N:Nat = s N ---> true
```


Soundness of induction proof

- **Sort** `Nat` is defined in `BASIC-NAT`, which is declared as **initial semantics** (`mod!`)
- Thus, for any model M denoted by `BASIC-NAT`, for any element *n* in M_{Nat}, there is a term *t* constructed from `0` and `s_` such that $M_t = n$ (no junk)
- `BASIC-NAT` is **protected**, so it holds for `NAT+` too
- Therefore, a proof of *P*(*t*) for all terms constructed from `0` and `s_` implies a proof of *P*(*t*) for all terms of Sort `Nat`

```
mod! BASIC-NAT{
  [Zero NzNat < Nat]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
}
```

```
mod! NAT+ {
  pr(BASIC-NAT) ...
  eq N + 0 = N .
  eq M + s N = s(M + N) .
}
```

Adding proved equations

- If you succeed in proving P(x) for x in a sort **S** by the induction and the sort **S** is declared in an initial module and the module is protected, then you can declare P(X:**S**) without changing the denotation of the specification

```
mod! NAT+ {  ...
  vars M N : Nat
  eq N + 0 = N .
  eq M + s N = s(M + N) .
}
```

```
open NAT+ + EQL
  red 0 + 0 = 0 .
  op n : -> Nat .
  eq 0 + n = n  .
```

Prove **0 + N = N** by induction

```
mod! NAT+' { ...
  vars M N : Nat
  eq N + 0 = N .
  eq M + s N = s(M + N) .
  eq 0 + N = N .
}
```

The denotations of `NAT+` and `NAT+'` are same

Example 2: Commutativity of +

- Prove the addition operator to be commutative
  - P(N) = “ ∀ M:Nat . M + N = N + M”
  - When you have more than one variables in a property to be proved, you choose one of the variables as a target of the induction
  - In this case M and N are symmetric

```
open NAT+ + EQL
  var M : Nat
  red M + 0 = 0 + M .          I.B.
  op n : -> Nat .
  eq M + n = n + M .           I.H.
  red M + s n = s n + M .      I.S.
close
```

Proof failed

- You do not always succeed in Induction Proof

```
open NAT+ + EQL
  var M : Nat
  red M + 0 = 0 + M .            I.B.
  op n : -> Nat .
  eq M + n = n + M .             I.H.
  red M + s n = s n + M .        I.S.
close
```

```
%NAT+ + EQL>   red M + 0 = 0 + M .
(M = (0 + M)):Bool
```

The I.B. reduction does not return true

Find lemma

- To make a proof complete, some lemma is needed
- A suitable lemma may or may not be found from the result of the failed reduction

```
%NAT+ + EQL>   red M + 0 = 0 + M .
(M = (0 + M)):Bool
```

We already proved this equation
(0 is the left-identity of +)

Adding proved lemma

- We can add a proved equation in a proof score

```
open NAT+ + EQL
  vars M N : Nat .
  eq 0 + N = N .              Lemma
  op n : -> Nat .
  red M + 0 = 0 + M .         I.B.
  eq M + n = n + M .          I.H.
  red M + s n = s n + M .     I.S.
close
```

```
%NAT+ + EQL> red M + 0 = 0 + M .
(true):Bool
%NAT+ + EQL> red M + s n = s n + M .
((s (n + M)) = ((s n) + M)):Bool
```

Thanks to Lemma, I.B. succeeds, but I.S. does not succeed

Complete proof

- The result of the reduction of the Induction Step also can be proved by the induction

```
%NAT+ + EQL> red M + s n = s n + M .
((s (n + M)) = ((s n) + M)):Bool
```

Proof of "s M + N = s(M + N)"

```
open NAT+ + EQL
 var M : Nat
 op n : -> Nat .
 red s M + 0 = s(M + 0) .        I.B.
 eq s M + n = s(M + n) .         I.H.
 red s M + s n = s(M + s n) .    I.S.
close
```

Proof of "M + N = N + M"

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

NAT+

```
eq N + 0 = N .
eq M + s N = s(M + N) .
```

All reductions return true, and The proof has been completed

Specification of lists

- The following parameterized module specifies lists whose elements can be of any kind of sets

```
hwd:mod* TRIV {
 [ Elt ]
}
```

```
mod! BASIC-LIST(X :: TRIV) {
  [Empty NeList < List]
  op nil : -> Empty
  op _::_ : Elt List -> NeList
}
```

```
view t2n from TRIV to NAT{ sort Elt -> Nat }
```

```
BASIC-LIST(X <= t2n)> parse 0 :: 1 :: 2 :: nil .
(0 :: (1 :: (2 :: nil))):NeList
```


Concatenation of lists

- Specify a concatenation function of lists

```
mod! LIST-@ {
  pr(BASIC-LIST)
  op _@_ : List List -> List
  var E : Elt .
  vars L1 L2 : List .
  eq       nil @ L1 = L1 .
  eq (E :: L1) @ L2 = E :: (L1 @ L2) .
}
```

```
LIST-@(X <= t2n)> red (0 :: 1 :: nil) @ (2 :: 3 :: nil) .

[1]: ((0 :: (1 :: nil)) @ (2 :: (3 :: nil)))
[2]: (0 :: ((1 :: nil) @ (2 :: (3 :: nil))) )
[3]: (0 :: (1 :: (nil @ (2 :: (3 :: nil))) ))
---> (0 :: (1 :: (2 :: (3 :: nil))))
```

Example 3: Associativity of @

- Prove the associativity of `@`
  - `(A @ B) @ C = A @ (B @ C)`
  - Which variable should we apply the induction to?

P(L) = (A @ B) @ L = A @ (B @ L)

```
open LIST-@ + EQL .
  vars A B : List
  op l : -> List .
  var E : Elt .
  red (A @ B) @ nil = A @ (B @ nil) .
  eq (A @ B) @ l = A @ (B @ l) .
  red (A @ B) @ (E :: l)
               = A @ (B @ (E :: l)) .
close
```

P(L) = (L @ B) @ C = L @ (B @ C)

```
open LIST-@ + EQL .
  vars B C : List
  op l : -> List .
  var E : Elt .
  red (nil @ B) @ C = nil @ (B @ C) .
  eq (l @ B) @ C = l @ (B @ C) .
  red ((E :: l) @ B) @ C
               = (E :: l) @ (B @ C) .
close
```

LIST-@

```
eq nil @ L1 = L1 .
eq (E :: L1) @ L2 = E :: (L1 @ L2) .
```


Adding associativity

- The commutative law and the associative law should be declared as an attribute of the operator
  - Remind of the lecture on the term rewriting system

```
mod! LIST-@-assoc {
  pr(LIST-@)
  eq (A:List @ B:List) @ C:List = A @ (B @ C) .
}
```

```
mod! LIST-@-assoc {
  pr(LIST-@)
  op _@_ : List List -> List {assoc}
}
```

Reverse function

- Specify a reverse function

```
mod! LIST-rev {
  pr(LIST-@-assoc)
  op rev _ : List -> List
  var E : Elt .  var L : List .
  eq rev nil = nil .
  eq rev (E :: L) = (rev L) @ (E :: nil) .
}
```

```
LIST-rev(X <= t2n)> red rev (0 :: 1 :: 2 :: nil) .
(2 :: (1 :: (0 :: nil))):NeList
```


`rev` is naive

- Computation of rev is not so efficient
  - For example, rev [0,1, ..., n-1, n] is first reduced into [n] @ [n-1] @ ... @ [1] @ [0] and then the concatenation of the singleton lists is reduced by the equation on @
  - Here, [0,1,...,n] is an abbreviation of (0 :: 1 :: ... :: n :: nil)

```
1:  rev [0, 1, 2]
2:  (rev [1, 2]) @ [0]
3:  (rev [2]) @ [1] @ [0]
4:  (rev nil) @ [2] @ [1] @ [0]
5:  nil @ [2] @ [1] @ [0]
6:  [2] @ [1] @ [0]
7:  2 :: (nil @ [1] @ [0])
8:  2 :: ([1] @ [0])
9:  2 :: 1 :: (nil @ [0])
10: 2 :: 1:: [0]
  = 2 :: 1 :: 0 :: nil
```

rev is $O(n^2)$

For an input list whose length is n, rev computes it by (n+1)(n+2) / 2 rewrite steps
(1 + 2 + ... + (n+1) + (n+2))

Smart reverse

- The following improved reverse function takes a list to be reversed and the empty list, and returns the reversed list
  - The reversed list is made one by one in the second argument of revi

```
mod! LIST-revi {
  pr(BASIC-LIST)
  op revi : List List -> List
  vars L1 L2 : List
  var E : Elt
  eq revi(nil, L2) = L2 .
  eq revi((E :: L1), L2) = revi(L1, (E :: L2)) .
}
```


Trace of revi

- For an input list whose length is n, the improved reverse computes it by n + 1 rewrite steps (O(n))

```
LIST-revi(X <= t2n)>
red revi(0 :: 1 :: 2 :: 3 :: 4 :: 5 :: nil ,nil)):List

[1]: revi( (0 :: (1 :: (2 :: (3 :: (4 :: (5 :: nil)))))) , nil )
[2]: revi( (1 :: (2 :: (3 :: (4 :: (5 :: nil))))) , (0 :: nil) )
[3]: revi( (2 :: (3 :: (4 :: (5 :: nil)))) , (1 :: (0 :: nil)) )
[4]: revi( (3 :: (4 :: (5 :: nil))) , (2 :: (1 :: (0 :: nil))) )
[5]: revi( (4 :: (5 :: nil)) , (3 :: (2 :: (1 :: (0 :: nil)))) )
[6]: revi( (5 :: nil) , (4 :: (3 :: (2 :: (1 :: (0 :: nil))))) )
[7]: revi( nil , (5 :: (4 :: (3 :: (2 :: (1 :: (0 :: nil)))))) )

(5 :: (4 :: (3 :: (2 :: (1 :: (0 :: nil)))))):NeList
(0.000 sec for parse, 7 rewrites(0.000 sec), 13 matches)
```

Example 4: `rev` = `revi`

- Prove that the naive reverse and the smart reverse denote a same function
  - `revi(L, nil) = rev L`

```
-- PROVE revi (L, nil) = rev L
open LIST-rev + LIST-revi + EQL .
 var E : Elt .
 op l : -> List .
 red revi(nil, nil) = rev nil .                I.B.
 eq revi(l, nil) = rev l .                     I.H.
 red revi(E :: l, nil) = rev (E ::             I.S.
l) .
close
```

```
%LIST-...> red revi(nil, nil) = rev nil .
(true):Bool
%LIST-...> red revi(E :: l, nil) = rev (E :: l)
(revi(l,(E :: nil)) = ((rev l) @ (E :: nil))):Bool
```


Lemma discovery failed

- If you take the result of the failed reduction itself as a lemma, it fails again and again

```
%LIST-...> red revi(E :: l, nil) = rev (E :: l)
(revi(l,(E :: nil)) = ((rev l) @ (E :: nil))):Bool
```

```
-- PROVE revi(l, E :: nil) = (rev l) @ (E :: nil)
open LIST-rev + LIST-revi + EQL .
 ...
  red revi(F :: l, E :: nil)= (rev (F :: l)) @ (E :: nil) .
close
```

```
%LIST-...> red ...
(revi(l,(F :: (E :: nil))) = ((rev l) @ (F :: (E :: nil)))):Bool
```

```
-- PROVE revi(l, F :: E :: nil) = (rev l) @ (F :: E :: nil)
open LIST-rev + LIST-revi + EQL .
 ...
 red revi(G :: l, F :: E :: nil)= (rev (G :: l)) @ (F :: E :: nil) .
close
```

```
 %LIST-...> red ...
(revi(l,(G :: (F :: (E :: nil)))) = ((rev l) @ (G :: (F :: (E :: nil))))):Bool
```

Generalization

```
revi(l, G :: F :: E :: nil) = (rev l) @ (G :: F :: E :: nil)
```

- From the result of the last reduction, you may think of a generalized equation
  - The second argument `G :: F :: E :: nil` of `revi` in lhs is same as the second argument of `@` in rhs
  - Thus, `revi(L, L')  = (rev L) @ L'` may hold, which is a generalization of `revi(L, nil) = rev L`

Prove a generalized equation

- The proof score of `revi(L, L') = (rev L) @ L'` returns true

```
open LIST-rev + LIST-revi + EQL .
 var E : Elt .
 var L' : List .
 op l : -> List .
 red (rev nil) @ L' = revi(nil, L') .
 eq revi(l, L')  = (rev l) @ L' .
 red (rev (E :: l)) @ L' = revi(E :: l, L') .
close
```

Proof completed

- Adding the lemma (the generalized equation) makes the proof score succeed

```
open LIST-rev + LIST-revi + EQL .
 var E : Elt .
 vars L L' : List
 eq revi(L, L')  = (rev L) @ L' .
 op l : -> List .
 red revi(nil, nil) = rev nil .
 eq revi(l, nil) = rev l .
 red revi(E :: l, nil) = rev (E :: l) .
close
```


Comments are important

- In this material, although we did not write comments in our examples because of a space limitation, comments are heavily recommended to be added to specifications and proof scores

```
-- Proof of revi(L, nil) = rev L
open LIST-rev + LIST-revi + EQL .
 -- Declare Lemma
 eq revi(L:List, L‘:List)  = (rev L) @ L' .
 -- Declare an arbitrary element for Induction
 op l : -> List .
 --> Prove Induction Base
 red revi(nil, nil) = rev nil .
 -- Declare Induction Hypothesis
 eq revi(l, nil) = rev l .
 --> Prove Induction Step
 red revi(E:Elt :: l, nil) = rev (E :: l) .
close
```

Summary

- Showed several induction proofs
- To make a proof score of induction
  – Decide which variable you apply the induction
  – Check the target sort is declared in an initial module and the module is protected
  – Declare a constant as an arbitrary element
  – Reduce I.B. for all constants of the sort
  – Declare I.H. by the constant declared as an arbitrary element
  – Reduce I.S. for all (non-constant) constructors of the sort
  – Find, prove, and declare lemma if the proof score failed
    - The result of reduction, or a generalization

TRS revised

- In the description stage with rapid-prototyping, we may obtain specifications which satisfy termination, confluence, etc
- In the verification stage, we may not obtain such good properties because equations (rewrite rules) are added in proof scores
  – Especially you may get non-confluence TRS
  – When getting non-confluence TRS, the precedence of equations may be important
    - Equations in the latest module are precedent
    - Upper equations are precedent in a same module