Transfinite Induction via Term Refinement in CafeOBJ

Kokichi Futatsugi

A transfinite induction method is formalized as a method for constructing proof scores in the CafeOBJ specification verification language system, and the effectiveness of the method is demonstrated with a couple of instructive examples.

Transfinite induction is induction in which a goal predicate can be assumed, as the induction hypotheses, to hold for all values smaller than the goal value with respect to a well-founded order.

A property of interest on a specification (i.e. a goal property) is described as a Boolean ground term `prp(c1,c2,..,cn)` with fresh constants `c1,c2,..,cn` which corresponts to the parameters of the goal property.

A specification is formalized as a set of equations each of those can be interepreted as a left to right reduction rule. The property proves to hold if the term `prp(c1,c2,..,cn)` is reduced to `true`.

Usually, term refinement of and induction on `c1,c2,..,cn` are necessary to show that `prp(c1,c2,..,cn)` is reduced to `true`. The transfinite induction in CafeOBJ naturally combines the term refinement of and the induction on `c1,c2,..,cn`, and has a potential to support a variety of inductions.

1 Introduction

CafeOBJ [2] is a state-of-the-art algebraic specification language and is a member of the OBJ language family [12] which also includes Maude [11]. CafeOBJ is designed as an executable specification language system which can be used as a specification verification system [9] [4] [5] [6] [13].

A CafeOBJ's specification is a description of models, and its verification is to show that the models has some desirable properties by deducing the properties from the specification. Specification verification in this sense is theorem proving where a set of axioms is the specification and theorems are the desirable properties.

The theorem proving capability of CafeOBJ has been significantly enhanced with some achievements including (1) formalization of induction and casesplit based on constructor-based specification calculus [8] and (2) specification/verification of transition systems with conditional transition rules [7] [13].

A proof score is a description in CafeOBJ of a proof. A specification in CafeOBJ is a set of equations, and CafeOBJ implements equational inference through reduction (i.e. rewriting with the equations by using them as left to right rewrite/reduction rules). For a CafeOBJ's specification (i.e. a specification module) M and a property p on M, expression $M \models p$ means that each model of M satisfies p. If CafeOBJ returns `true`

CafeOBJ における項詳細化による超限帰納法

二木厚吉, 産業技術総合研究所/国立情報学研究所/北陸先端科学技術大学院大学, National Institute of Advanced Industial Science and Technology/National Institute of Informatics/Japan Advanced Institute of Science and Technology.

for a reduction '`reduce in` M : p .' (i.e. p reduces to `true` at module M) then $M \vDash p$ is proved. A reduction that returns true is called effective. A proof score for $M \vDash p$ is a description of a set of effective reductions $\{$'`reduce in` M_i : p_i .' $\mid i = 1, 2, \cdots, n\}$ (including descriptions of modules that are necessary to execute the reductions) such that

($M_1 \vDash p_1$ `and` $M_2 \vDash p_2$ `and` $\cdots$ `and` $M_n \vDash p_n$) `implies` $M \vDash p$

Proof scores can naturally describe several kinds of inductions including structural induction [1] [3], and quite a few cases are developed until now [5] [6] [13]. Transfinite induction seems to subsume all of inductions used and could have a potential to enhance effectiveness and efficiency of proof scores.

2 An Explanatory Example

Simple CafeOBJ specifications and proof scores are presented, and several fundamental concepts and techniques necessary for constructing proof scores with transfinite induction are explained.

2.1 Specification of _+_ on Peano Natural Numbers

The following CafeOBJ module `PNAT` (`04:-08:`) specifies Peano nutural numbers, and module `PNAT+` (`12:-17:`) specifies a binary plus operator `_+_` on the natural numbers.

Line number like `07:` is for explanation and is not part of CafeOBJ code, and CafeOBJ code starts from the 4th column in each line.

Lines starting with `--` or `-->` are comments (`01:-03:`, `09:-11:`)

Two constructor operators `0` and `s_` are declared (`06:-07:`) and sort `Pnat` (`05:`) is defined as a set of terms of the form $\overbrace{\texttt{s s} \cdots \texttt{s}}^{n}$ `0` ($n \in \{0, 1, 2, \cdots\}$).

$$\texttt{Pnat} = \{\texttt{0}, \texttt{s 0}, \texttt{s s 0}, \texttt{s s s 0}, \cdots\}$$

`14:` declares the rank (i.e. arity and co-arity) of the binary plus operator `_+_`, and `15:-16:` defines its function (or meaning).

```
01: --> ---------------------------------------
02: --> PNAT: Peano NATural numbers
03: --> ---------------------------------------
04: mod! PNAT {
05: [Pnat]
06: op 0 : -> Pnat {constr} .
07: op s_ : Pnat -> Pnat {constr} .
08: }
09: --> ---------------------------------------
10: --> PNAT with the addition _+_
11: --> ---------------------------------------
12: mod! PNAT+ {
13: pr(PNAT)
14: op _+_ : Pnat Pnat -> Pnat .
15: eq 0 + Y:Pnat = Y .
16: eq s X:Pnat + Y:Pnat = s(X + Y) .
17: }
```

2.2 Proof Score with Ordinary Induction

The following CafeOBJ code (`01:-40:`) is a proof score for proving that the equation `[assoc+]` (`04:-06:`) holds for any elements `x, y, z` of sort `Pnat`. That is, the binary operator `_+_` satisfies the associative law.

`ASSOC+` (`10:-19:`) is a module for writing the proof score. It imports the base module `PNAT+` (`12:`) on which the property going to be proved (associativity in this case) applies.

Any property described with equations (like `04:-06:`) can be expressed with a Boolean ground term (a term without variables) with fresh constants (as `14:+16-18:`), and showing that the Boolean ground term (`assoc+` in this case) reduces to `true` is sufficient for proving the property (see `Proposition` 3.2).

```
01: --> ======================================
02: --> proof score for proving
03: --> associativity of _+_:
04: -->  eq[assoc+]:
05: -->    (X:Pnat + Y:Pnat) + Z:Pnat =
06: -->    X + (Y + Z) .
07: --> with the induction on X:Pnat
08: --> ======================================
09: --> module for declaring the goal of proof
10: mod ASSOC+ {
11: -- the base module
12: pr(PNAT+)
13: -- Boolean ground term expressing the goal
14: pred assoc+ : .
15: -- fresh constants
```

```
16: ops x y z : -> Pnat .
17: eq assoc+ =
18:       (((x + y) + z) = (x + (y + z))) .
19: }
```

A CafeOBJ command 'red in ASSOC+ : assoc+ .' does not return true, and the proof does not succeed. A common technique to proceed with the proof is **case splitting**. x is a fresh constant (i.e. a constant never used in the context) of sort Pnat, and, for another fresh constant x', two equations 'eq x = 0 .' (23:) and 'eq x = s x' .' (32:) covers all possible cases. The specification of Pnat, 0, and s_ at 2. 1/05:-07: justifies this.

'open ASSOC+ .' (21:) opens module ASSOC+ and create a tentative module at which all sorts, operators, and equation of ASSOC+ are available. At this tentative module, after declaring 'eq x = 0 .' (23:) the reduction command 'red assoc+ .' (25:) returns true. This implies the proof is done for the case where 'eq x = 0 .' (23:) holds, and 20:-26 is a correct and effective proof score for the induction base case on x.

For the case where 'eq x = s x' .' (32:) holds, if the induction hypothesis on x' is declared (34:), the reduction command 'red assoc+ .' (36:) returns true. This implies that the proof is done for the case where (32:) and (34:) hold, and 27:-37 is a correct and effective proof score for the induction step case on x.

```
20: --> induction base on x
21: open ASSOC+ .
22: -- base case
23: eq x = 0 .
24: -- check the base
25: red assoc+ . --> succeed
26: close
27: --> induction step on x
28: open ASSOC+ .
29: -- fresh constant for refining x
30: op x' : -> Pnat .
31: -- step case
32: eq x = s x' .
33: -- induction hypothesis
34: eq (x' + Y:Pnat) + Z:Pnat = x' + (Y + Z) .
35: -- check the step
36: red assoc+ . --> succeed
37: close
38: --> QED
39: --> ====================================
40: --> ====================================
```

Comment expressed in 38:-40: indicates that 09:-37: is correct and effective proof score for proving the equation 04:-06: in the comment 01:-08: if 25: and 36: return true.

2. 3 Proof Score with Transfinite Induction

The following 01:-72: is another proof score for proving the associativity equation [assoc+] (11:-13:). It uses transfinite induction, and 47:-60: and 61:-70: is two different pieces of code with a same function for executing reductions for the proof. That is, either of 17:-60: or 17:-46:+61:-70: is a correct and effective proof score for the equation [assoc+] (04:-06:).

Module NAT< declared the fact 06: on the built-in greater-than predicate _<_ on built-in sort Nat of built-in module NAT that is going to be used in the proof score.

27:-39: defines the induction order _<io_ (32:) on 3 tuples (X:Pnat Y:Pnat Z:Pnat) of three parameters (variables) of the equation [assoc+] (11:-13:). Note that the order _<io_ is irreflexive and well-founded.

40:-43: defines the induction hypothesis by a conditional equation with a condition composed of the induction order _<io_. Note that this conditional equation is effective for any 3 tuple (X:Pnat Y:Pnat Z:Pnat) that satisfies the condition '(X Y Z) <io (x y z)', and constitutes induction hypotheses of transfinite induction.

Because _<io_ is well-founded, if assoc+ is reduces to true at module ASSOC+ind that includes the induction hypothesis 40:-43:, then **transfinite induction principle** (**Proposition** 3.2) guarantees that the goal equation 11:-13: holds for any 3 tuple (X:Pnat Y:Pnat Z:Pnat).

```
01: --> ------------------------------------
```

```
02: --> built-in NAT with FTBU on _<_
03: --> ---------------------------------------
04: mod! NAT< {
05: pr(NAT)
06: eq N1:Nat < NZ:NzNat + N1 = true .
07: }
08: --> =======================================
09: --> proof score for proving
10: --> associativity of _+_:
11: -->  eq[assoc+]:
12: -->    (X:Pnat + Y:Pnat) + Z:Pnat =
13: -->    X + (Y + Z) .
14: --> with the transfinite induction
15: --> on 3 tuple (X:Pnat Y:Pnat Z:Pnat)
16: --> =======================================
17: -- module for proof with transfinite induction
18: mod ASSOC+ind {
19: -- base module
20: pr(PNAT+)
21: -- goal Boolean ground term
22: pred assoc+ : .
23: -- fresh constants
24: ops x y z : -> Pnat .
25: eq assoc+ =
26:        (x + (y + z) = (x + y) + z) .
27: -- induction order
28: -- on 3 tuple (X:Pnat Y:Pnat Z:Pnat)
29: -- which is well founded order
30: [Pnat3tpl] op ___ : Pnat Pnat Pnat ->
31:                      Pnat3tpl {constr} .
32: pred _<io_ : Pnat3tpl Pnat3tpl .
33: pr(NAT<)
34: op sz : Pnat -> Nat .
35: eq sz(0) = 0 .
36: eq sz(s N:Pnat) = 1 + sz(N) .
37: eq (X1:Pnat Y:Pnat Z:Pnat) <io
38:    (X2:Pnat Y:Pnat Z:Pnat) =
39:    sz(X1) < sz(X2) .
40: -- induction hypothesis
41: cq (X:Pnat + Y:Pnat) + Z:Pnat =
42:    X + (Y + Z)
43:    if (X Y Z) <io (x y z) .
44: -- fresh constant for refinement
45: op x' : -> Pnat
46: }
```

'`eq x = 0 .`'(`52:`) and '`eq x = s x' .`'(`57:`) covers all possible cases, and if both of '`red assoc+ .`' at `53:` and `58:` return `true`, then the proof succeeds. The two reductions return `true`, and `17:-60:` is a correct and effective proof score for `[assoc+]` (`11:-13:`).

`61:-70:` execute the proof similarly as `48:-59:` by using the CafeOBJ's specification calculus facility. `63:` select the base module where the proof is executed. `64:` declares that '`eq assoc+ = true .`' is the goal equation, and create a tentative goal module called the root which import the base module and include the goal equation. `65:-66:` defines that `x` is a proof rule for splitting the current goal (case) into two goals (cases) that include '`eq x = 0 .`'(`65:`) or '`eq x = s x' .`'(`66:`).

`67:` applies the proof rules `x` and `rd` successively to the current goal. Applying `x` creates two subgoals (i.e. two tentative modules) by adding each of the two equations to the current goal module respectively. Applying `rd` after `x` does reductions at each of two sub-goals and check whether the goal equation holds. If the goal equation holds, the goal is discharged. If all sub-goals of a goal is discharged, the goal is dischaged, and the proof succeeds if the root goal is discharged.

```
47: --> =======================================
48: --> proof score with open..close
49: -- case1 and case2 cover all possibilities
50: -- case1: eq x = 0 .
51: open ASSOC+ind .
52: eq x = 0 .
53: red assoc+ .
54: close
55: -- case2: eq x = s x' .
56: open ASSOC+ind .
57: eq x = s x' .
58: red assoc+ .
59: close
60: --> QED
61: --> =======================================
62: --> proof score with SpecCalc
63: select ASSOC+ind .
64: :goal{eq assoc+ = true .}
65: :def x = :csp{eq x = 0 .
66:               eq x = s x' .}
67: :apply(x rd)
68: :show proof
69: :desc proof
70: --> QED
71: --> =======================================
72: --> =======================================
```

'`:apply(x rd)`'(`67:`) discharges all the created goals and the proof succeeds. `68:` and `69:` are commands for inspecting proofs. `73:-75:` is output for `68:`, and `78:-93:` is output for `69:`. `73:` shows the root goal, and `74:-75:` shows two subgoals `1` and `2` created by the proof rule `x`. `*` after a goal in `73:-75:` shows that the goal is discharged. `78:-93:` describe the proof in more detail, and shows direct correspondence between `47:-60:` and

```
61:-70: clearly.

73: root*
74: [x]   1*
75: [x]   2*
76:
78: ==> root*
79:     -- context module: #Goal-root
80:     -- targeted sentence:
81:       eq assoc+ = true .
82: [x]     1*
83:     -- context module: #Goal-1
84:     -- assumption
85:       eq [x]: x = 0 .
86:     -- discharged sentence:
87:       eq [RD]: assoc+ = true .
88: [x]     2*
89:     -- context module: #Goal-2
90:     -- assumption
91:       eq [x]: x = s x' .
92:     -- discharged sentence:
93:       eq [RD]: assoc+ = true .
```

3 Basics of Transfinite Induction via Term Refinement

This section presents basics of transfinite induction via term refinement in a modestly formal way.

3. 1 Transfinite Induction

Let Srt be a set of elements with a **well founded** binary relation $\_<_{io}\_$. That is, there is no infinite sequence of elements $e_1, e_2, e_3, \cdots$ such that $e_{i+1} <_{io} e_i$ $(i \in \{1, 2, 3 \cdots\})$.

The principle of transfinite induction is stated as follows.

Proposition 3.1 [Transfinite Induction]
Let $p(X : Srt)$ be a predicate with a parameter of sort Srt.

$$(\forall b \in Srt((\forall a <_{io} b(p(a))) \texttt{ implies } p(b))) \\ \texttt{implies } (\forall c \in Srt(p(c)))$$

□

3. 2 Theorm of Constants

In CafeOBJ, a property of interest on a module M is supposed to be described as a Boolean term composed of operators available at the module M. The Boolean term is expressed as $\texttt{prp}(v_1, \cdots, v_n)$, has $n \in \{0, 1, 2, \cdots\}$ parameters (or variables) $v_1{:}\texttt{Sort}_1, \cdots, v_n{:}\texttt{Sort}_n$ of specific sorts, and the property of interest is $(\forall v_1{:}\texttt{Sort}_1, \cdots, \forall v_n{:}\texttt{Sort}_n\ (\texttt{prp}(v_1, \cdots, v_n)))$ (i.e. the parameters are universally quantified). Let $\texttt{c}_1{:}\texttt{Sort}_1, \cdots, \texttt{c}_n{:}\texttt{Sort}_n$ be n fresh constants which correspond to the n parameters $v_1{:}\texttt{Sort}_1, \cdots, v_n{:}\texttt{Sort}_n$. We get the following proposition.

Proposition 3.2 [Theorm of constants]
$(\forall v_1{:}\texttt{Sort}_1, \cdots, \forall v_n{:}\texttt{Sort}_n(\texttt{prp}(v_1, \cdots, v_n)))$ holds if $\texttt{prp}(\texttt{c}_1, \cdots, \texttt{c}_n)$ reduces to `true`.

□

The original **"Theorm of constants"** in [10] states the definitional equivalence of the variables $v_1{:}\texttt{Sort}_1, \cdots, v_n{:}\texttt{Sort}_n$ and the fresh constants $\texttt{c}_1{:}\texttt{Sort}_1, \cdots, \texttt{c}_n{:}\texttt{Sort}_n$ in defining algebraic models and says nothing about reduction.

3. 3 Casesplit with Equations

A module consists of a set St of sorts, a set Op of operations, and a set Eq of equations declared in the module. A model of the module is a system that consists of the following two entities that satisfy all equations in Eq. (1) Sets which correspond to St. (2) Operations on the sets which correspond to Op.

Let M be a module and p be a property on M with universally quantified parameters, and let $M \models p$ mean that p holds for any model of M. As explained in 3. 2, the property p can be assumed to be expressed as a Boolean ground term. Let $M \vdash_{\mathrm{r}} b$ mean that a Boolean ground term b reduces to `true` at module M. The following restates **Proposition 3.2** by fixing the module of interest.

$$M \vdash_{\mathrm{r}} p \texttt{ implies } M \models p \qquad (1)$$

Let $e_1, \cdots, e_n$ be equations such that at least one of them holds for any possible case. That is, the equations cover all the possibilities. Usually n is 2 or 3. Let M_{+e_i} be a module gotten by adding e_i to M, then each model of M is a model of M_{+e_j} for some $j \in \{1, 2, \cdots, n\}$, and the following (2) states

the principle of **casesplit with equations**.

$$(M_{+e_1} \models p \texttt{ and } M_{+e_2} \models p \texttt{ and } \cdots \texttt{ and } M_{+e_n} \models p) \texttt{ implies } M \models p \quad (2)$$

3.4 Term Refinement with Equations

A fresh constant is declared to be of some sort, and can be refined with equations which cover all posibilites.

2.1/`05:-07:` specifies sort `Pnat` and operators `0`, `s_` as follows.

```
[Pnat]
op 0 : -> Pnat {constr} .
op s_ : Pnat -> Pnat {constr} .
```

Based on the above specification, a fresh constant delcared as '`op n : -> Pnat .`' can be refined into two possible cases with two equations '`eq n = 0 .`' and '`eq n = s` $\mathtt{n}_1$ `.`', where $\mathtt{n}_1$ is another fresh constant of sort `Pnat` (i.e. is delcared as '`op` $\mathtt{n}_1$ `: -> Pnat .`').

$\mathtt{n}_1$ can be refined with similar two equations in turn, and `n` can be refined into a sufficiently detailed term '`s s ...` $\mathtt{s}\ \mathtt{n}_m$' of sort `Pnat`.

3.5 Specification Calculus

CafeOBJ code like 2.3/`63:-67:` is a proof score with specification calculus (a SpecCalc proof score for short). Specification calculus is going to be formalized as operations on proof trees caused by a SpecCalc proof score. As a result, the SpecCalc proof score is shown to present a semi-algorithm for proving $M \models p$ for a property p expressed as a Boolean ground term on a module M.

For defining proof trees, nodes of a proof tree are expressed as follows. `root` denotes the root node of a proof tree. n-`1`,n-`2`...n-k ($k \in \{1, 2, ...\}$) are child nodes of node n, and n is the mother (or father) node of n-j ($j \in \{1, 2, ..., k\}$). A set T of nodes is called a **tree** if n is an element of T then the mother of n is an element of T. A node in a tree is called **leaf** if it has no child node.

A node `root-`n is abbreviated to n, and any node except `root` can be considered to be a sequence of non-zero natural numbers. Then the lexicographical order can be defined over nodes except `root`; `root` is assumed to be the first in the lexicographical order.

A symbol `>` is put to a node for indicating the next node of a proof tree.

A proof tree is a tree structure of modules. That is, each node of the proof tree is a module for proving $M_i \models p_i$ for some M_i and p_i, and the module imports M_i and has equation '`eq` p_i `= true .`' as the targeted sentence. The node/module is called a **goal** $M_i \models p_i$. If $M_i \vdash_{\mathrm{r}} p_i$ the node/module/goal is discharged, and a symbol `*` is put to the node.

[Create the `root` Node] –

The following CafeOBJ code creates a module with a node `root` for proving $M \models p$.

```
select M .
:goal{eq p = true .}
```

The current proof tree consists only of the node `root`. Put `>` to the node `root` (i.e. the goal $M \models p$), and make it the `next goal`.

[Define Casesplit Equation Lists] –

The following CafeOBJ code defines lists of equations $eqs_1, eqs_2, \cdots, eqs_l$ each list of which $eqs_i = eq_{i1}\, eq_{i2} \cdots eq_{im_i}$ ($i \in \{1, 2, \cdots, l\}$) covers all prossible cases, and can be used for casesplit. Each of $eqs_1, eqs_2, \cdots, eqs_l$ is called a **caseslpit eq-list**.

```
:def eqs1 = :csp{eq11 eq12 ··· eq1m1}
:def eqs2 = :csp{eq21 eq22 ··· eq2m2}
    ...
:def eqsl = :csp{eql1 eql2 ··· eqlml}
```

[Apply sequence of `:apply` commands] –

Let γ denote a sequence each element of which is a caseslpit eq-list eqs_i or `rd-`, e.g.

$$\gamma = eqs_1\ \texttt{rd-}\ eqs_2\ \texttt{rd-}\ eqs_m\ \texttt{rd-}.$$

Let t denote a proof tree and let n denote a node in the proof tree, and let `:apply(`γ`)@`n`[`t`]` denote the proof tree after applying `:apply(`γ_1`)` to t at n.

The proof tree after applying a sequence of

`:apply` commands

`:apply(`γ_1`) :apply(`γ_2`)` $\cdots$ `:apply(`γ_m`)`

to the current proof tree ct is defined as follows.

```
:apply(γm)@nm−1
  [:apply(γm−1)@nm−2
    ···
      [:apply(γ2)@n1
        [:apply(γ1)@n0[ct]]···]
```

Here n_0 is the next node of the current proof tree and n_i is the next node of the proof tree after applying `:apply(`γ_i`)`

[Apply `:apply` command] –

For a node n that does not have child nodes and is not discarged (i.e. to which `*` is not put) in a proof tree t, `:apply(`γ`)@`n`[`t`]` is defined recursively as follows.

[If γ is empty] –

`:apply(`γ`)@`n`[`t`]` $= t$.

[If γ is not empty] –

Let $\gamma = r\,\gamma_r$ for $r = eqs$ or $r =$ `rd-`, and let the node n be a goal $Mm \vDash pp$.

[If $r = eqs$] –

If $eqs = eq_1\, eq_2 \cdots eq_m$, then create m child nodes n`-`j of n for each $j \in \{1, 2, \cdots, m\}$ such that the node n`-`j is a goal $M_{+eq_j} \vDash p$. `:apply(`$eqs\,\gamma_r$`)@`n`[`t`]` is defined as follows.

```
:apply(eqs γr)@n[t] =
  :apply(γr)@n-m
    [apply(γr)@n-(m−1)
      [···
        [:apply(γr)@n-2
          [:apply(γr)@n-1[t]]···]
```

If `>` is put to the node n (i.e. n is the next node), then delete `>` from n and put it to the node n`-1`, i.e. n`-1` is the new next node.

[If $r =$ `rd-`] –

[If $Mm \vdash_{\mathrm{r}} pp$ holds] –

Put `*` to the node n to indicate the node is discharged. For each node in the current proof tree, put `*` to it if each of its child nodes has `*` (i.e. is discharged). If n is the next node, delete `>` from n. Put `>` to the nearest next non-discharged leaf node (i.e. a leaf node without `*`) in the lexicographic order if such a node exits.

[If $Mm \vdash_{\mathrm{r}} pp$ does not hold] –

`:apply(rd-`γ_r`)@`n`[`t`] =`
`:apply(`γ_r`)@`n`[`t`]`

The **restatement of theorem of constants** (1) and the **principle of casesplit with equations** (2) of 3. 3 justifies the following propostion.

Proposition 3.3 [SpecCalc]

Let an initial proof tree consist only of the node `root`, which is a goal $M \vDash p$ and is the next goal (i.e. to which `>` is put). If the application of a sequence of `:apply` commands:

`:apply(`γ_1`) :apply(`γ_2`)` $\cdots$ `:apply(`γ_m`)`
$(m \in \{1, 2, \cdots\})$

terminates and each node in the resultant proof tree is discharged (i.e. to which `*` is put), then $M \vDash p$ is proved. □

4 conclusion

Transfinite induction has been formaized as a method for constructing proof scores in CafeOBJ. The formalized proof score method has the following characteristics.

- Induction hypotheses are directly declared with conditional equations (i.e. conditional reduction-rules), and a wide varaiety of induction hypotheses could be defined in an executable way.
- Casesplit with equations for searching all the possible candidates can be applied uniformly to the module in which induction hypotheses are declared, and the principle of transfinite induction stated at the proposition 3.2:

  $(\forall b \in Srt((\forall a <_{io} b(p(a)))$ `implies` $p(b)))$
  `implies` $(\forall c \in Srt(p(c)))$

  is coded almost directly into the CafeOBJ code

like 2.3/41:-43,63:-67:. 2.3/63:-67: corresponds to the outer universal quantification $\forall b \in Srt$, and 2.3/41:-43 corresponds to the inner universal quantification $\forall a <_{io} b$.

- Term refinement with equations is nicely combined with casesplit with equations and realizes **casesplit with term refinement**. The casesplit with term refinement lies in the core of specification calculus, and could make the full use of transfinite induction hypotheses defined in conditional equations.

References

[1] Burstall, R.: Proving properties of programs by structural induction, *Computer Journal*, Vol. 12(1)(1969), pp. 41–48.

[2] CafeOBJ: Web page, http://cafeobj.org/, 2018.

[3] Diaconescu, R.: Structural induction in institutions, *Inf. Comput.*, Vol. 209, No. 9(2011), pp. 1197–1222.

[4] Diaconescu, R. and Futatsugi, K.: *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, AMAST Series in Computing, Vol. 6, World Scientific, 1998.

[5] Futatsugi, K.: Verifying Specifications with Proof Scores in CafeOBJ, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, IEEE Computer Society, 2006, pp. 3–10.

[6] Futatsugi, K.: Fostering Proof Scores in CafeOBJ, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, Dong, J. S. and Zhu, H.(eds.), Lecture Notes in Computer Science, Vol. 6447, Springer, 2010, pp. 1–20.

[7] Futatsugi, K.: Generate & Check Method for Verifying Transition Systems in CafeOBJ, *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, Nicola, R. D. and Hennicker, R.(eds.), Lecture Notes in Computer Science, Vol. 8950, Springer, 2015, pp. 171–192.

[8] Futatsugi, K., Gâinâ, D., and Ogata, K.: Principles of proof scores in CafeOBJ, *Theor. Comput. Sci.*, Vol. 464(2012), pp. 90–112.

[9] Futatsugi, K. and Nakagawa, A. T.: An Overview of CAFE Specification Environment - An Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications over Networks, *First IEEE International Conference on Formal Engineering Methods, ICFEM 1997, Hiroshima, Japan, November 12-14, 1997, Proceedings*, IEEE Computer Society, 1997, pp. 170–182.

[10] Goguen, J.: *Theorem Proving and Algebra*, [Unpublished Book Draft].

[11] Maude: Web page, http://maude.cs.uiuc.edu/, 2018.

[12] The OBJ Family: Web page, http://cseweb.ucsd.edu/~goguen/sys/obj, 2018.

[13] Kokichi Futatsugi (二木厚吉): *Introduction to Specification Verification in CafeOBJ (*モデルの記述と検証のためのプログラミング入門-*CafeOBJ* による仕様検証-*)*, Saiensu-Sha (サイエンス社), 2017.

A Another Simple Example: Total Order _<_ on Pnat

```
--> ---------------------------------------
--> PNAT with equality _=_
--> ---------------------------------------
mod! PNAT=
{
pr(PNAT)
op _=_ : Pnat Pnat -> Bool {comm} .
eq ((s M:Pnat) = 0) = false .
eq (s M:Pnat = s N:Pnat) = (M = N) .
}
--> ---------------------------------------
--> PNAT with greater predicate _<_
--> ---------------------------------------
mod! PNAT<
{
pr(PNAT=)
op _<_ : Pnat Pnat -> Bool .
eq 0 < s N:Pnat = true .
eq M:Pnat < 0 = false .
eq (s M:Pnat < s N:Pnat) = M < N .
}
--> =======================================
--> proof score for proving
--> eq[ex<>=]:
-->   (M:Pnat < N:Pnat) xor
-->   (N < M) xor (M = N) xor
-->   ((M < N) and (N < M) and (M = N))
-->   = true .
--> at module PNAT<
--> with transfinite induction on 2 tuple
--> (M:Pnat N:Pnat)
--> via term refinement
--> =======================================
-- module for proof
-- with transfinite induction
mod EX<>=ind {
-- base module
pr(PNAT<)
-- goal proposition
pred ex<>= : .
ops m n : -> Pnat . -- fresh constants
eq ex<>= =
    (m < n) xor (n < m) xor (m = n) xor
     ((m < n) and (n < m) and (m = n)) .
-- induction order
```

```
-- on 2 tuples of Pnat (M:Pnat N:Pnat)
-- which is well founded order
[Pnat2tpl] op __ : Pnat Pnat ->
                   Pnat2tpl {constr} .
pred _<io_ : Pnat2tpl Pnat2tpl .
eq (M:Pnat N:Pnat) <io (s M s N) = true .
-- induction hypothesis
-- for complete induction
cq (M:Pnat < N:Pnat) xor
   (N < M) xor (M = N) xor
   ((M < N) and (N < M) and (M = N))
   = true
   if (M N) <io (m n) .
-- fresh constants for refinement
ops m' n' : -> Pnat .
}
--> ======================================
-- specCalc proof score
select EX<>=ind .
:goal{eq ex<>= = true .}
:def m = :csp{eq m = 0 . eq m = s m' .}
:def n = :csp{eq n = 0 . eq n = s n' .}
:apply(m n rd) -- succeed
--
:show proof
:desc proof
**> QED [ex<>=]
--> ======================================
--> ======================================

#| -- comments start
-- out put for :show proof
root*
[m]   1*
[n]   1-1*
[n]   1-2*
[m]   2*
[n]   2-1*
[n]   2-2*

-- out put for :disc proof
==> root*
    -- context module: #Goal-root
    -- targeted sentence:
      eq ex<>= = true .
[m]     1*
    -- context module: #Goal-1
    -- assumption
      eq [m]: m = 0 .
    -- targeted sentence:
      eq ex<>= = true .
[n]     1-1*
    -- context module: #Goal-1-1
    -- assumptions
      eq [m]: m = 0 .
      eq [n]: n = 0 .
    -- discharged sentence:
      eq [RD]: ex<>= = true .
[n]     1-2*
    -- context module: #Goal-1-2
    -- assumptions
      eq [m]: m = 0 .
      eq [n]: n = s n' .
    -- discharged sentence:
      eq [RD]: ex<>= = true .
[m]     2*
    -- context module: #Goal-2
    -- assumption
      eq [m]: m = s m' .
    -- targeted sentence:
      eq ex<>= = true .
[n]     2-1*
    -- context module: #Goal-2-1
    -- assumptions
      eq [m]: m = s m' .
      eq [n]: n = 0 .
    -- discharged sentence:
      eq [RD]: ex<>= = true .
[n]     2-2*
    -- context module: #Goal-2-2
    -- assumptions
      eq [m]: m = s m' .
      eq [n]: n = s n' .
    -- discharged sentence:
      eq [RD]: ex<>= = true .

-- comments end |#
```