数理解析研究所講究録
第2008巻 2016年 37-46

The proof structure of a proof assistant Isabelle/HOL

Hidetsune Kobayashi (Institute of Computational Logic)
Yoko Ono (Yokohama City University)

1 Intoduction

In Isabelle/HOL, when a proposition is given, we can try to prove it by applying propositions stored within Isabelle/HOL[1]. Almost all proof steps are backward inference, that is "for the conclusion to hold, we ask, which conditions should be statisfied". The number of conditions to be satisfied is not necessarily one, so continueing several backward inference steps, the number of conditions may become bigger and bigger. But, in our exprience, we do not have so many conditions to check, provided the original proposition is not so complicated.

Therefore we have fortunate cases in which all steps can be given mechanically. However, if a proposition contains some complicated mathematical properties, we cannot give a complete proof by this step by step mechanical method, since it has no grand design.

In this report, taking the Bernstein's theorem[3] as an example, we see how our automated reasoning system "HPR" works, and we see how the complete proof in Isabelle/HOL is given partially by HPR.

2 Proof steps

Since Isabelle/HOL stores proved propositions within Isabelle database, it is not necessary to use database system outside. But it is still useful to store the propositions within an ourside database system. We use postgreSQL as a database, in which propositions, types of propositions and some mathematical knowledge are stored.

To choose appropriate propositions to make a proof of a new proposition:

1. look for propositions within DB having similar conclusion to the conclusion of the proposition to prove.

2. among selected propositions, we compare premises to omit not suitable propositions
3. substitute values of the proposition to prove for variables of the selected propositions
4. ask Isabelle/HOL whether a selected proposition with replaced values can be applied successfully or not. If not, try another.
5. If no applicable proposition from the DB, we return the comment "Have no idea" and wait for an input by a user.

We are developing an automated reasoning system H-prover which is based on Isabelle 2011 and using ProofGeneral as an interface. Since ProofGeneral use emacs and its source code is written in emacs lisp, so our automated reasoning system is written in emacs lisp and postgreSQL [2] is combined together with ProofGeneral and Isabelle. In our automated reasoning system, propositions are expressed in trees obtained by automatic conversion, therefore above steps are executed in emacs and SQL database system. In emacs codes are written in emacs lisp and in postgreSQL, codes are written in lisp like language.

3 Propositions in Data base

To avoid confusion, we call "rules" propositions already proved and stored within a DB, and we call "proposition" a proposition to prove.

As stated in the previous section, we express propositions in tree. We have two kinds of trees, one is a binary tree in which the root has left child and right child, and another is a linear tree in which the root has only one child. In Isabelle/HOL the variables of a rule is marked with preceding ? like ?x. Those variables preceded by ? should be replaced by corresponding values in the proposition(to prove).

Here we show some tipical trees:

```
binary (= (?f ?x) (?g ?x))      ((?f ?'a RarS ?'b)
                                  (?g ?'a RarS ?'b) (?x ?'a))
       (inS (?c) (?B))          ((?c ?'a) (?B ?'a RarS bool))
linear (?P ?b)                  ((?P ?'a RarS bool) (?b ?'a))
       (exS! $x dS = ($x) (?t)) (($x ?'a) (?t ?'a))
```

Above `(= (?f ?x) (?g ?x))` is represented as `?f ?x = ?g ?x` in Proof-General with X-symbol, and `(inS (?c) (?B))` as `?c ∈ ?B`, `(exS! $x dS = ($x) (?t))` as `∃x. x = t`. In Isabelle, each variable has its type which is listed on the right hand of a tree.

Rules in elementary set theory are contained in files HOL.thy, Set.thy and FuncSet.thy which are written by L. Paulson and others. The total number of axioms, definitions and propositions are 678.

Among the rules, the conclusion of which are binary_trees are 523 and that of linear trees beginning with q_variables are 64, and that of linear trees beginning with reserved symbols 43 and others 48. Reserved symbols appearing at the head of conclusions are following 15:

= (394)	inS (50)	sueS (40)	lrarS (15)	eqvS (14)
neqS (10)	orS (6)	andS (4)	sbS (4)	ninS (3)
intS (1)	unS (1)	leqS (1)	llrarS (1)	

where the number in brackets denotes the number of binary trees with the root. We see the number of conclusions biginning with '=' is 394, therefore if the conclusion of the proposition with '=', we have to select apropreate rules among these 394 rules. Among these 394, having an atom at the first letter of the left child is 187. Atoms appearing more than 10 times are − (26), ?f (10), SetS (14), bqS (12), intS (27), invimsS (13), unS (24). Among tree having '=' at the head of the tree and 'intS' at the head of the left child, the atom appearing at the head of the right child are − (2), ?A (3), ?B (2), SetS (6), insert (3), intS (7), unS (2).

Above numbers imply that if we compare head of conclusion, head of left child, head of right child of rules with that of proposition, we can narrow down to select candidate rules to apply.

But we have a cumbersome propositions with '=' at the head of the conclusion tree and both left and right child begin with q_variables:

conclusion	types
(= (?t) (?t))	((?t ?'a))
(= (?f ?x) (?g ?x))	((?f ?'a RarS ?'b) (?g ?'a RarS ?'b) (?x ?'a))
(= (?f) (?g))	((x ?'a) (?f ?'a RarS ?'b) (?g ?'a RarS ?'b))
(= (?f ?x) (?f ?y))	((?x ?'a) (?y ?'a) (?f ?'a RarS ?'b))
(= (?f ?a ?c) (?f ?b ?d))	((?a ?'a) (?b ?'a) (?c ?'b) (?d ?'b) (?f ?'a RarS ?'b RarS ?'c))
(= (?f ?x) (?g ?y))	((?f ?'a RarS ?'b) (?g ?'a RarS ?'b) (?x ?'a) (?y ?'a))

```
(= (?P) (?Q))          ((?P bool) (?Q bool))
(= (?f) (?f))          ((?f ?'a RarS ?'b))
(= (?t) (?t))          ((?t ?'a))
(= (?A) (?B))          ((?A ?'a RarS bool) (?B ?'a RarS bool))
(= (?t) (?s))          ((?s ?'a) (?t ?'a))
(= (?r) (?t))          ((?r ?'a) (?s ?'a) (?t ?'a))
(= (?b) (?a))          ((?b ?'a) (?a ?'a))
(= (?r) (?t))          ((?r ?'a) (?s ?'a) (?t ?'a))
(= (?x) (?y))          ((?x ?'a) (?y ?'a))
(= (?A) (?B))          ((?A ?'a) (?B ?'a))
(= (?x) (?y))          ((?x ?'a) (?y ?'a))
(= (?a) (?b))          ((?a ?'a) (?b ?'a))
(= (?c) (?d))          ((?a ?'a) (?b ?'a) (?c ?'a) (?d ?'a))
(= (?A) (?B))          ((x ?'a) (?A ?'a RarS bool)
                                   (?B ?'a RarS bool))
(= (?f) (?g))          ((?f ?'a RarS ?'b) (?A ?'a RarS bool)
                                     (?g ?'a RarS ?'b) (x ?'a))
(21 rows)
```

Among above trees, following 11 have similar types "`('a)`", `(= (?t) (?t))`, `(= (?t) (?t))`, `(= (?t) (?s))`, `(= (?r) (?t))`, `(= (?b) (?a))`, `(= (?r) (?t))`, `(= (?x) (?y))`, `(= (?A) (?B))`, `(= (?x) (?y))`, `(= (?a) (?b))`, `(= (?c) (?d))`, so comparing conditions of a rule and the proposition, there are some cases types can show distinction, but even checking types there are some which cannot be seen to be distinct or not. In the last case, we have to check the premises of the proposition and the rule.

Here similar types, in the simplest case, is like this: types v and v1 is similar if both

```
v similar to '\(\??''[a-zA-Z][a-zA-Z0-9]*\)'
v1 similar to '\(\??''[a-zA-Z][a-zA-Z0-9]*\)'
```

are true, where the right-hand side of each line is the regular expression (Cf. [2]) That is, if both v and v1 are similar to the same regular expression of types, then v and v1 are similar types. In the regular expression, \?? means with one preceding ? or no ?, therefore even if v and v1 are similar, these two two are not necessarily the same.

Thus if the conclusion of the proposition is a binary tree then we can narrow down candidate rules only by comparing conclusions except some special cases.

If the conclusion of the proposition is expressed by a simple linear tree, we cannot narrow down candidate only by comparing the conclusions.

```
select name, premises, conclusion, get_type0(conclusion, types)
       from propositions where not binary_treep(conclusion)
       and slength(conclusion)=1 and q_variablep(car(conclusion))
       order by no;
```

```
name             premises                     conclusion   get_type0
__________       _______________              _____        _____

mp               ((lrarS (?P) (?Q)) (?P))     (?Q)         (bool)
iffD2            ((= (?P) (?Q)) (?Q))         (?P)         (bool)
rev_iffD2        ((?Q) (= (?P) (?Q)))         (?P)         (bool)
iffD1            ((= (?Q) (?P)) (?Q))         (?P)         (bool)
rev_iffD1        ((?Q) (= (?Q) (?P)))         (?P)         (bool)
iffE             ((= (?P) (?Q)) (...))        (?R)         (bool)
eqTrueE          ((= (?P) (True)))            (?P)         (bool)
allE             ((falS $x dS ?P $x) ··· )    (?R)         (bool)
all_dupE         ((falS $x dS ?P $x) ··· )    (?R)         (bool)
FalseE           ((False))                    (?P)         (bool)
False_neq_True   ((= (False) (True)))         (?P)         (bool)
notE             ((ntS ?P) (?P))              (?R)         (bool)
...              ...                          ...          ...
(54 rows)
```

Here the premises are represented by lists of trees. For example the first one `((lrarS (?P) (?Q)) (?P))` consists of two trees `(lrarS (?P) (?Q))` and `(?P)`. The rules appearing in the table TABLE, are fundamental propositions concerning logic, and by checking premises, we can obtain a few candidate rules.

We present next simple case:

```
select conclusion, get_type0(car(conclusion), types),
       get_type0(cadr(conclusion), types) from propositions
       where
       not binary_treep(conclusion) and
       slength(conclusion)=2 and
       q_variablep(car(conclusion)) and
       q_variablep(cadr(conclusion));
```

```
conclusion         get_type0              get_type0
--------           ---------              ---
(?P ?t)            (?'a RarS bool)        (?'a)
(?f ?s)            (?'a RarS ?'b)         (?'a)
(?P ?t)            (?'a RarS bool)        (?'a)
(?P ?a)            (?'a RarS bool)        (?'a)
(?P ?b)            (?'a RarS bool)        (?'a)
(?P ?x)            (?'a RarS bool)        (?'a)
(?P ?a)            (?'a RarS bool)        (?'a)
(?P ?x)            (?'a RarS bool)        (?'a)
(?P ?x)            (bool RarS bool)       (bool)
(9 rows)
```

The rule, with the conclusion represented as the first tree `(?P ?t)` with types `(?P ?'a RarS bool)` and `(?t ?'a)` respectively, can be applied to the conclusion `(andS (P x) (Q x))` with type `((P 'a RarS bool) (Q 'a RarS bool) (x ('a)))`. The last tree can be rewritten as `((lmbS $x dS and (P $x) (Q $x)) x)`. In such case, we check whether the conclusion of the proposition (in this case `(andS (P x) (Q x))`, has a variable with type similar to that of `?P` or not. After comparing `(?P ?t)` and `(andS (P x) (Q x))`, we substitute `(lmbS $x dS and (P $x) (Q $x))` for `?P` and substitute `x` for `?t`.

Note that to compare expressions of propositions, we have to calculate types of expressions. For example, let f be a function of type ('a RarS 'b), g be a function of type ('b RarS 'c) and x be of type ('a), then $(g \circ f)$ x and g (f x) should be the same element. After type calculation, we see that these two have the same type ('c). This is a necessary condition that these two are the same.

4 Lemmas, definitions and a primitive recursion for the Bernstein's theorem

Now we focus on the Bernstein's theorem concerning the cardinality of infinite sets. It is impossible to complete the proof with propositions within the data base. (Note that within DB only propositions in HOL.thy, Set.thy and FuncSet.thy).

We have totally 43 new propositions for the Bernstein's theorem. It is expressed by tree as

```
(llrarS (lrBRK (inS (f) (rarS (A) (B)) sclS inj_on f A
sclS inS (g) (rarS (B) (A)) sclS inj_on g B)) (exS $h dS
andS (inS ($h) (rarS (A) (B))) (bij_to $h A B)))
```

In the interface ProofGeneral with X-symbols, this is converted as

$[$f $\in$ A $\rightarrow$ B; inj_on f A; g $\in$ B $\rightarrow$ A; inj_on g B $]\Longrightarrow$
$\exists$h. h $\in$ A $\rightarrow$ B $\wedge$ bij_to h A B.

This theorem is reduced to a simpler case:

lemma BernsteinTr3:"$[$A1 $\subseteq$ A; f $\in$ A $\rightarrow$ A1; inj_on f A $]\Longrightarrow$
$\exists$ g. g $\in$ A $\rightarrow$ A1 $\wedge$ bij_to g A A1"

The key idea is, by using iteration, make a function g of the last lemma.

```
primrec itr ::"[nat, 'a ⇒ 'a] ⇒ ('a ⇒'a)" where
    itr_0 :  "itr 0 f = f" |
    itr_Suc:  "itr (Suc n) f = comp f (itr n f)"

definition A2set::"['a ⇒'a, 'a set, 'a set] ⇒ 'a set" where
    "A2set f A A1 == {x.  x ∈ A1 ∧(∃y ∈(A - A1).  ∃n.  itr n f
y = x)}"

definition Bfunc::"['a ⇒ 'a, 'a set, 'a set] ⇒ ('a ⇒ 'a)" where
    "Bfunc f A A1 == λx∈A. if (x ∈(A - A1) ∪ (A2set f A A1)) then
f x else x"
```

The key idea of the proof of Bernstein's theorem is to introduce a set A2set f A A1 and introduce a function Bfunc f A A1 : A $\rightarrow$ A1 which is a bijective map. And this shows the existence of g in the lemma BernsteinTr

To show that Bfunc f A A1 : A $\rightarrow$ A1 is injective, we use auxiliary lemmas as follows:

name	contents
`Bfunc_fun`	Bfunc is a function from A to A1
`Bfunc_inj_1`	a preparatory lemma to show `Bfunc f A A1` is injective
`Bfunc_inj`	Shows `Bfunc` is injective. A lemma injective_iff is required

To show that Bfunc f A A1 : A $\rightarrow$ A1 is surjective, we require following lemmas:

name	contents
Elem_fixed	required to show Bfunc_surj
Bfunc_surj	Shows Bfunc is surjective. We require lemmas

name	contents
surj_to_test	to test whether surjective
Bfunc_fun	to see Bfunc is a function
BernsteinTr2	show Bfunc is surj_to
A2_set_sub	show A2 set is a subset of A1
Bfunc_eq	to show Bfunc is a function
Elem_fixed	required to see Bfunc is a function
Bfunc_id	required to see Bfunc is surjective

We show whole steps of a proof to the lemma "Bfunc_surj" (the steps are NOT automatically generated).

```
lemma Bfunc_surj:"⟦A1 ⊆ A; f ∈ A → A1; inj_on f A⟧
                                      ⟹ surj_to (Bfunc f A A1) A A1"
```

`apply (rule surj_to_test)`
Now have two subgoals:

```
  1.  ⟦A1 ⊆ A; f ∈ A → A1; inj_on f A⟧⟹ Bfunc f A A1 ∈ A → A1
  2.⟦A1 ⊆ A; f ∈ A → A1; inj_on f A⟧⟹ ∀ b∈A1.  ∃a∈A. Bfunc f
A A1 a = b
```

`apply (rule Bfunc_fun, assumption+)`
This resolves the first subgoal. Now, the subgoal 2 becomes the subgoal 1.

`apply (rule ballI)`
This is a common way to rewrite the subgoal.

`apply (case_tac "b ∈ A2set f A A1")`
This is one of the key points. How to find out this automatically? We have two subgoals. case 1 b ∈ A2set f A A1 and case 2 b ∉ A2set f A A1.

`apply (frule BernsteinTr2[of A1 A f], assumption+)`
Puts surj_to f (A - A1 ∪ A2set f A A1) (A2set f A A1) within premises.

`apply (simp only:surj_to_def)`
Rewrite the new assumption.

```
apply (drule_tac t = "f ` (A - A1 ∪ A2set f A A1)" and s = "A2set
f A A1" and P = "λxxx.  b ∈ xxx" in ssubst, assumption+)
```

drule_tac adds a new assumption, drule_tac removes some redundant premises. After execution drule_tac, we cannot use removed one. Frule also adds a new condition into premises, without removing.

apply (drule_tac b = b and f = f and A = "A - A1 ∪ A2set f A A1" in mem_in_image)
Now subgoals are:

1. ⋀b.⟦A1 ⊆ A; f ∈ A → A1; inj_on f A; b ∈ A1; b ∈ A2set f A A1; ∃a∈A - A1 ∪ A2set f A A1. b = f a⟧⟹ ∃a∈A. Bfunc f A A1 a = b
2. ⋀b.⟦A1 ⊆ A; f ∈ A → A1; inj_on f A; b ∈ A1; b ∉ A2set f A A1⟧⟹ ∃a∈A. Bfunc f A A1 a = b

apply (erule_tac bexE)
This rewrites the subgoal 1 as

⋀b a. ⟦A1 ⊆ A; f ∈ A → A1; inj_on f A; b∈ A1; b *in* A2set f A A1; a ∈ A - A1 ∪ A2set f A A1; b = f a⟧⟹ ∃a∈A. Bfunc f A A1 a = b

apply (rule_tac x = a in bexI)
Try "a" as an element satisfying the conclusion.

apply (subst Bfunc_eq, assumption+)
Substitute Bfunc f A A1 a = f a in the conclusion.

apply (rule sym, assumption)
Rewrite f a = b as b = f a.

apply (frule_tac a = a in A2set_sub[of A1 A f], assumption+)
Puts a ∈ A into premises. The subgoal 1 is resolved.

apply (frule_tac a = a in Bfunc_id[of A1 A f], assumption+)
Puts Bfunc f A A1 b = b into premises.

apply simp
Resolves
1. ⋀b. ⟦A1 ⊆ A; f ∈ A → A1; inj_on f A; b ∈ A1; b ∉ A2set f A A1 ⟧⟹ b ∉ A - A1 ∪ A2set f A A1

apply (rule subsetD[of A1 A], assumption+)
b ∈ A1 and A1 ⊆ A then b ∈ A. Now remaining subgoal is

1. ⋀b. ⟦A1 ⊆ A; f ∈ A → A1; inj_on f A; b ∈ A1; b ∉ A2set

```
f A A1 ⟧⟹ ∃a∈A. Bfunc f A A1 a = b.
```

```
apply (rule_tac x = b in bexI, assumption+)
```
Try “b”.

```
apply (rule subsetD[of A1 A], assumption+)
done
```

References

[1] T. Nipkow, L. Paulson and M. Wenzel, *Isabelle/HOL: A Proof Assitant for Higher Order Logic* (Springer, 2010)

[2] *PostgreSQL 9.1.1 Documentation* (http://www.postgresql.org/)

[3] S. Kametani, *Set and Topology* (Asakura Shoten, 2004)