数理解析研究所講究録
第1964巻 2015年 35-39

An automated reasoning system with a preparatory inference

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

1 Introduction

Our automated reasoning system "H-prover" uses Isabelle/HOL[1] as an inference engine. To prove a proposition, Isabelle has four tactics rule_tac, drule, erule and frule_tac. The rule_tac rewrites a proposition by applying a theorem or a lemma under a condition. That is, except some theorems, rule_tac works only if two conditions are satisfied:

1. the conclusion of the theorem coincides the conclusion of the proposition to be proved with a proper substitution of variables
2. the first premise is satisfied.

Therefore if, by checking the conclusion part, a theorem seems to be used to rewrite the proposition to proved, but the theorem's first premise is not satisfied by premises of the proposition, then we try to derive the first premise of the theorem from premises of the proposition to prove by using a frule_tac.

In this report, we present a method:

1. to choose a theorem applied by using rule_tac.
2. to derive the first premise of the theorem from premises of the proposition to prove.

As noted above a tactic used in the second step is called a "frule_tac" in Isabelle.

We note that in Isabelle each variable has a type, such as "'a", "'a ⇒ bool" and "'a ⇒ 'b". At first we illustrate a proof procedure:

1. give a proposition to be proved into emacs
2. ProofGeneral transfers it to Isabelle
3. Isabelle gives a response
4. ProofGeneral transfers them to emacs
5. H-prover takes the response and choose some theorems from a SQL table which can be used to rewrite the proposition
6. H-prover gives one among chosen theorems to Isabelle and check whether is works or not
7. if a theorem works, repeats from 4 to 6

2 Extracting a character of a theorem

Given a proposition to be proved, H-prover chooses some candidate theorems used to rewrite the proposition from a SQL table. In the table each theorems is stored as a tree.

In this section, we discuss a character of a theorem which enables us to choose proper theorems applied to rewrite the proposition to prove. At first we give two examples:

```
axiom mp:"⟦ ?P ⟶ ?Q; ?P ⟧ ⟹ ?Q"
```

The tree expression of the axiom mp is

```
(llrarS (lrBRK (lrarS (?P) (?Q) sclS ?P)) (?Q))
```

and both ?P and ?Q have type "bool". This axiom has two premises `(lrarS (?P) (?Q))` and `(?P)` and the conclusion is `(?Q)`. `?P` and `?Q` are sbstituted by some variables or formulae when this axiom is used to rewrite a proposition. Using a regular expression (of postgreSQL), we take skeletons of

```
(?Q)                as  \(_+\)
(lrarS (?P) (?Q))   as  \(lrarS \(_+\) \(_+\)\)
```

```
lemma exI:" ?P ?x ⟹ ∃ y.  ?P y"
```

The tree expression of this lemma is

```
(llrarS (?P ?x) (exS $x dS ?P $x))
```

The type of ?P is "?'a ⇒ bool", the type of ?x is "?'a" and the type of $x is "?'a". Skeleton are

```
(?P ?x)             as  \(_+ _+\)
(exS $x dS ?P $x)   as  \(exS \$[^ ]+ dS _+ \$[^ ]+\)
```

Thus the skeleton of a proposition tree is obtained by changing a local variable as $[^] and changing the other variables as _+. In SQL we choose candidate theorems as

> select name from rule_table where `conclusion1` similar to skeleton(`conclusion`)

where `conclusion1` is the conclusion of the proposition to prove and `conclusion` is the conclusion of a theorem in the table. If the conclusion is simple as in the axiom mp, since any tree is similar to \(_+\), any theorem with the same type conclusion is selected.

Positions of variables are another character of a proposition. For the tree `(?P ?x)`, the position of ?P is (_e) and that of ?x is (_c _e). Here (_e) means root position, and (_c _e) means root position of the child. Since a variable, say ?P, sometimes occur at different positions, we express positions as

$$\texttt{((?P position}_1\ \texttt{position}_2\ \ldots\ \texttt{position}_n\texttt{) (?Q positions) ...)}$$

Since the branch point of a proposition tree, if any, is binary, a position of a variable is specified by the shortest path from the root to the variable. For example the position of the variable ?A in the tree `(andS (?A) (?B))` is (_l _e) and that of ?B is (_r _e). "_l" means the left-child, and "_r" means the right-child. We have one more sign to express a position within extra brackets as in the following example:

$$(\exists x{\in}?A.\ ?P) = ((\exists x.\ \ x \in ?A) \wedge ?P)$$

The tree expression of the above is
`(= ((exS inS ($x dS ?P) (?A))) ((andS ((exS $x dS inS ($x) (?A))) (?P))))`
The position of the variable ?P is (_l _n _c _l _c _c _e). Thus the position of a variable can be expressed by _e, _c, _l, _r and _n. Note that the following example shows simply taking propositions makes an error.

$$\texttt{proposition xxx:"[\![P c; Q c]\!]} \Longrightarrow \exists\ \texttt{x.}\ \ \texttt{P x} \wedge \texttt{Q x""}$$

This proposition is rewritten by the lemma exI, but the conclusion of exI is ∃ y. ?P y. The position of _P is (_c _c _c _e), but at this point "andS" is located in the tree of the proposition to prove. This is because ?P in exI is an operator, and using an operator $\exists\ \texttt{x.}\ \ \texttt{P x} \wedge \texttt{Q x}$ should be expressed as $\exists\texttt{\$x.}\ \ \lambda\texttt{\$x.}\ \ \texttt{P \$x} \wedge \texttt{Q \$x x}$. Thus an operator should be rewritten by using λ expression.

We note the type of a tree. We can calculate a type of a tree. We illustrate how to calculate the type by giving a simple tree (?f ?x), where the type of ?f is ?'a ⇒ ?'b and the type of ?x is ?'b. The type of (?f ?x) is ?'b since the type of (?f ?x) is the type of the image element. Like this we can calculate the type of a tree.

We saw items that characterize a proposition

1. skeleton of the proposition
2. types of variables
3. positions of variables

If the skeleton of the conclusion of a theorem is similar to that of a proposition to prove, we check whether the type of variables (and trees) are the same or not. Finally checking positions of each variable of the theorem coincide that of the proposition to prove, we can obtain variables to substitute the variables of the theorem.

3 Deriving the first premise

Given a proposition to prove, we choose a theorem as above from the rule_table omitting the first premise of the theorem. Then we have two cases:

case 1 the first premise of the theorem is satisfied

case 2 the first premise of the theorem is not satisfied

In the case 1, we have only to use a rule_tac with specified variables. In the case 2

1. checking positions, we specify variables of the theorem (we call theorem1 the new theorem with substituted variables)
2. choose a theorem which derives the first premise of theorem1
3. applying frule_tac, we rewrite theorem1
4. apply the theorem chosen in 1 by using rule_tac to theorem1

We show how tactics work by giving examples:

`prop_to prove injective:"[| inj_on f A; x ∈ A; y ∈ A; f x = f y |] ⟹ x = y "`
`apply (unfold inj_on_def)` *expand the definition gives the following*

`"[| ∀x ∈ A. ∀y ∈ A. f x = f y ⟶ x = y; x ∈ A; y ∈ A |] ⟹ f x = f y"`

We use the following theorem bspec and add `f x = f y ⟶ x = y` *as an extra premise of the proposition to prove.*

`bspec: "[| ∀x ∈ ?A. ?P ?x; ?x ∈ A |] ⟹ ?P ?x" as`

`apply (frule_tac x = x in bspec)` *then Isabelle returns two subgoals*

`1. "[| ∀x ∈ A. ∀y ∈ A. f x = f y ⟶ x = y; x ∈ A; y ∈ A; f x = f y |] ⟹ x ∈ A";`

2. "⟦ ∀x ∈ A. ∀y ∈ A. f x = f y ⟶ x = y; x ∈ A; y ∈ A; f x = f y; ∀y∈A. f x = f y ⟶ x = y ⟧ ⟹ x = y "

the subgoal 1 is trivial because the conclusion appears within premises. Applying bspec again to the subgoal 2, we obtain following two propositions.

1. "⟦ ∀y∈A. f x = f y ⟶ x = y; ∀x∈A. ∀y∈A. f x = f y ⟶ x = y; x ∈ A; y ∈ A; f x = f y ⟧ ⟹ y ∈ A"
2. "⟦ ∀y∈A. f x = f y ⟶ x = y; ∀x∈A. ∀y∈A. f x = f y ⟶ x = y; x ∈ A; y ∈A; f x = f y; f x = f y ⟶ x = y ⟧ ⟹ x = y"

Now we can apply rev_mp:"⟦ ?P; ?P ⟶ ?Q ⟧ ⟹ ?Q", with substitutions ?P ↪ f x = f y, ?Q ↪ x = y.

In this example, specialization is the straight forward way because of

∀x ∈ A. ∀y ∈ A. f x = f y ⟶ x = y; x ∈ A

so frule_tac can be applied without foresight. That is "even we don't know rev_mp is applied later, we apply frule_tac bspec".

But there is an example an frule_tac is applied effectively with a scope of complete proof.

mp_contra1: "⟦ P ⟶ ¬Q; Q ⟧ ⟹ ¬ P"

For this proposition, neither

`rule_tac P = P and Q = "¬ Q" in mp`

nor `notE` do not work. To this proposition, at first, we apply

`frule_tac P = P and Q = "¬ Q"` Then we can apply `notE` to complete a proof.

References

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

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