数理解析研究所講究録
第1915巻 2014年 26-37

A characteristic function to select rules for an automated prover

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

1 Introduction

An automated prover generates a sequence of steps rewriting a proposition by using an axiom or proved proposition, and the result of the final step is an apparently true proposition. Hereafter, to avoid confusion, we call a proposition to be rewritten as proposition, and we call a proposition used to rewrite as a rule. In general, there are some rules employed to rewrite a proposition and some of them results in failure. Therefore choosing proper rules from a database determines the efficiency of the automated prover.

An automated prover should add a newely proved proposition to the database as a rule, hence a function used to choose proper rules should also decide whether the new rule is applicable or not. In other words, we cannot remake a rule selection function after each time we prove a new proposition.

In this report, we give a selection method with extracting a character of a rule.

2 Extracting a character of a proposition

Since our objective is to make an automated prover for Bernstein's theorem[1] in set theory, the scope of the function extracting a character of propositions is limited to elementary set theory.

Our prover consists of three parts, Isabelle/HOL[2], ProofGeneral and PostgreSQL. Isabelle/HOL is the inference engine, ProofGeneral is the interface and PostgreSQL is the database of the prover, and these these three parts are connected by using LISP language. A proposition is expressed in

logical expression in the interface, but those expressions are expressed as trees in list of LISP. Therefore the function extracts a character of tree. In Isabelle, there are four ways to apply rules to rewrite a proposition:

```
rule xxx, drule xxx, erule xxx, frule xxx,
```

where, `xxx` is the name of a rule. In this report, we focus on `rule xxx` or `rule_tac xxx`.

Here we give an example to show how `rule xxx` is used to rewrite.

```
Proposition Example0:"⟦ P c; Q c⟧ ⟹ ∃ x.  P x ∧ Q x"
```

This proposition is changed to a simple form by using a rule

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

Substituting λy. P y ∧ Q y for ?P, we obtain the conclusion part of exI changed as ∃ y. P y ∧ Q y which is same as the conclusion part of Example0. By this substitution, ?P ?x is changed to P ?x ∧ Q ?x, so giving a special value c to ?x. If the proposition

```
⟦ P c; Q c ⟧ ⟹ P c ∧ Q c
```

is proved then Example0 is proved. Thus `apply (rule xxx)` use a rule `xxx` having the conclusion fit to the conclusion of the proposition. Note that Isabelle has `apply (rule_tac ... in xxx)` in which local variables can be specified. In Example0,

```
apply (rule_tac ?P = λy.  P y ∧ Q y in exI)
```

works. `rule_tac` specifies some variables, it can diminish ambiguity.

From now on, we discuss a function extracting a character of a proposition. A type is set to each variable in Isabelle, we see a value to asign is restricted according to types. Therefore the type of a variable is a part of the character of a proposition, but it is not sufficient to distinguish rules.

We present an example to show how to extract a character of a proposition.

```
lemma Example1:  g :  A → B, f :  B → C, x ∈ A ⟹ (f
· g) x ∈ C
```

Here (f·g) x is an element transformed by the composed function of g and f. The proposition states that if g is a function from A to B, and if f is a function from B to C, and x is an element of A, then (f·g) xis an element of C. To prove this lemma, at first, by using the definition of composition, we change (f·g) x to f(g x), then we apply the lemma called funcset_mem:

lemma funcset_mem: ⟦ ?f : ?A → ?B, ?x ∈ ?A ⟧ ⟹ ?f ?x ∈ ?B

Since we express a proposition in a tree, the conclusion parts of Example1, and funcset_mem are (inS ((circS (f) (g)) x) (C)) and (inS (?f ?x) (?B)) respectively.

Expressing each variable as ?, try (inS ((circS (?) (?)) ?)) and (inS (? ?) (?)) as charcters. To see automatically Example can be rewritten by the rule funcset_mem, the prover should see the conclusion of Example1 is the same as the conclusion of funcset_mem. Therefore expressing (circS (?) (?)) as ? we take the character of (inS ((circS (f) (g)) x) (C)) as (inS (? ?) (?)) or using % which matches any character in SQL, we take the character as (inS (% %) (%)). The character of (circS (f) (g)) is not always ?. That is, when it appears (circS (?) (?)) alone, we have to take the character as (circS (?) (?)).

Thus the character of a proposition should be decided by comparing with rules to apply. The following table shows characters of trees.

conclusion tree	characrer
(inS (x) (B))	(inS (?) (?))
(inS (f x) (B))	(inS (? ?) (?))
(inS ((circS (f) (g)) x) (C))	(inS (? ?) (?))
(inS (f x) (bqS (f) (B)))	(inS (? ?) (?))

The last expression of the table is a tree expression of (f x) ∈ f ‘ B, where f ‘ B is the image of a set B by the function f. For elementary rules, we have the following table.

(andS (A) (B))	(andS (?) (?))
(orS (A) (B))	(orS (?) (?))
(andS (P) (andS (Q) (R)))	(andS (?) (andS (?) (?))

Three expressions f ‘ A,{y. ∃ x ∈ A. y = f x}, {a} express sets. The first two are the same set.

If these appear alone or the front depth of the list expressing a tree is 1, characters are

(bqS (f) (A))	(bsS (?) (?))
(SetS y dS exS inS ($s dS = (y) (f $x)))	(SetS ? dS %)
(SetS a)	(SetS ?)

As noted above, rules are stored within postgreSQL, a character (SetS y dS %) is used to select rules to apply as:

```
select name from propositions where conclusion like '(SetS
? dS %)'
```

and so, % matches any pattern.

The following table presents names of rules selected from the SQL table "propositions" comparing the character of the conclusion of the following lemma

```
lemma func_to_img:"f ∈ A → f ‘ A "
```

.

name
funcsetI
func_to_img
bij_betw_imp_funcset
image_eqI
rev_image_eqI
surj_to_hom
subsetD
(7 rows)

Among these rules, funcsetI is the proper rule to apply. We apply the rule as

```
apply (rule funcsetI)
```

then, Isabelle returns proof (prove): step 1
goal (1 subgoal):
1. ⋀ x. x ∈ A ⟹ f x ∈ f ‘ A

Repeating selection in SQL as

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char1_c1);
```

```
name
-------------------------
rangeI
Pi_mem
extensional_funcset_mem
imageI
surj_to_mem
subsetD
(6 rows)
```

All of these are not suitable to apply, since the prover does not know what is the meaning of f ' A at this time. This implies we cannot select a proper rule by merely comparing characters. In this case, prover should unfold the definition of an image. So, we give the definition as

```
apply (subst image_def)
```

then, we have

```
proof (prove): step 2
goal (1 subgoal):
1. ⋀x. x ∈ A ⟹ f x ∈ {y. ∃x∈A. y = f x}
```

We present another example. A proposition to prove is:

```
lemma itr_into_A1:"⟦A1 ⊆ A; f ∈ A → A1; y ∈ A ⟧ ⟹ itr
n f y ∈ A1"
```

We try to select rules to apply as:

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

The result of this search is 0 row. Actually, we use a command

```
apply (induct_tac n)
```

The induct_tac is one of the tactics of proof in Isabelle, as rule_tac. And the table "propositions" is a table containing rules to be applied by `rule_tac`. Hence it is not possible to find out `induct_tac`. Now, we are discussing to select `XXX` of `apply (rule_tac XXX)`. Where`XXX` is the name of a rule.

Applying `induct_tac`, we have

```
proof (prove): step 1
goal (2 subgoals):
1. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A⟧ ⟹ itr 0 f y ∈ A1
2. ⋀n. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A; itr n f y ∈ A1⟧ ⟹ itr
(Suc n) f y ∈ A1
```

Since within the function compare1, used in SQL command selecting rules, we can add a function to find `induct_tac`,

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

gives Using this result, we put `apply(subst itr_0)` as an Isabelle command,

```
 name
------
 itr_0
 (1 row)
```

then we have

```
proof (prove): step 2
goal (2 subgoals):
1. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A⟧ ⟹ f y ∈ A1
2. ⋀n.⟦A1 ⊆ A; f ∈ A → A1; y → A; itr n f y ∈ A1⟧ ⟹ itr (Suc
n) f y ∈ A1
```

Again, using the function compare1,

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

We have Among these rules, funcset_mem is a proper rule to apply. Our prover generates automatically the following command:

```
apply (rule_tac f = f and A = A and B = A1 and x = y in
funcset_mem)
```

```
 name
------------------------
 extensional_funcset_mem
 imageI
 surj_to_mem
 funcset_mem
 (4 rows)
```

Applying the command, we have

```
proof (prove): step 3
goal (3 subgoals):
1. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A⟧ ⟹ f ∈ A → A1
2. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A⟧ ⟹ y ∈ A
3. ⋀n. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A; itr n f y ∈ A1⟧ ⟹ itr (Suc
n) f y ∈ A1
```

Among these three, 1 and 2 have conclusion within premise, the command apply asasumption+ sweaps away these two, and we have

```
proof (prove): step 4
goal (1 subgoal):
1. ⋀n. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A; itr n f y ∈ A1 ⟧ ⟹ itr (Suc
n) f y ∈ A1
```

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

```
 name
-------
 itr_Suc
 (1 row)
```

As above, we give the following command:

```
apply (subst itr_suc)
proof (prove): step 5
goal (1 subgoal):
1. ⋀n.
⟦A1 ⊆ A; f ∈ A → A1; y ∈ A; itr n f y ∈ A1⟧ ⟹ (f · itr n f)
y ∈ A1
```

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

```
 name
------
(0 row)
```

Because the propver does not know the definition of composed function.
So we give `apply (subst comp_def)` then we have:

```
proof (prove): step 6
goal (1 subgoal):
1. ⋀n. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A; itr n f y ∈ A1⟧ ⟹ f (itr
n f y) ∈ A1
```

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

```
 name
-----------
funcset_mem
(1 row)
```

Applying this rule, we have

```
proof (prove): step 7
goal (1 subgoal):
1. ⋀n. ⟦A1 ⊆ A; f ∈ A → A1; y ∈ A; itr n f y ∈ A1⟧ ⟹ itr n
f y ∈ A,
```

then we try choose next rule to apply:

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

```
 name
------
(0 row)
```

This failure is caused by the character `(inS (?) (?))` of `subsetD` "`(inS (?x) (?C))`" is too restrictive. If we take the character as `(inS (%) (%))`, then we can obtain subsetD as follows.

```
select name from propositions, prop_to_prove where compare1(name,
char2_c, char2_c1, char3_c1);
```

But, `(inS (%) (%))` matches patterns (inS (xxx) (yyy)) with any xxx and yyy, `subsetD` is often included as a superfluous one.

```
name
-------
subsetD
(1 row)
```

3 Checking premises

In section1, we selected rules by comparing the characters of conclusions. There are many propositions conclusions of them have the same character. Therefore we can expect that characters of premises distinguish rules. For example, following two different propositions have conclusions with the same character.

$$\texttt{trans} : \text{"}\texttt{a} = \texttt{b}, \texttt{b} = \texttt{c} \implies \texttt{a} = \texttt{c}\text{"}$$
$$\texttt{sym} : \text{"}\texttt{a} = \texttt{b} \implies \texttt{b} = \texttt{a}\text{"}$$

Represent their conclusions in tree as

`((= (a) (b)) (= (b) (c)))` and `((= (a) (b)))`

respectively. By ignoring difference of variables, we have

`((= (?) (?)) (= (?) (?)))` and `((= (?) (?)))`.

If sym is applicable to a proposition, then the character of the proposition's conclusion should be `(= (?) (?))` and the proposition should have a premise with a character `(= (?) (?))`. So, trans is also selected as a rewriting rule for the proposition. But if a proposition is rewritten by trans, then a rule to apply should have at least two premises, sym is not selected. However, comparing the numbers expressions within a premise makes a mistake as following examples show.

Example1: $[\![\ \texttt{P c, Q c}\ ]\!] \implies \exists \texttt{x}.\ \ \texttt{P x} \wedge \texttt{Q x}$

To this proposition, we apply

`exI: P c ==> ∃x. P x`

with P substituted by $\lambda \texttt{x}.\ \ \texttt{P x} \wedge \texttt{Q x}$. In this case `exI` has only one premise $\texttt{P c} \wedge \texttt{Q c}$ which is equivalent to $[\![\ \texttt{P c; Q c}\ ]\!]$.

In general, rule tactic is not applicable unless the first expression of the premise is satisfied by a proposition to prove. So, we compare a rule and a proposition to prove focusing on the relation between variables of the first expression in the premise and variables of the conclusion.

4 Other proof tactics: drule, erule, frule

We discussed Isabelle's tactic rule or rule_tac. Here we discuss briefly on drule,erule, frule.

drule: As an example, we give the following proposition.

```
Example_d:  b ∈ {y.∃x ∈ A.y = f x} ⟹ ∃a ∈ A.b = f a
```

To rewrite this proposition, we use the rule

CollectD: a ∈ {x. ?P x} ⟹ ?P ?a

as `apply (drule CollectD)`. Then we have:

∃x ∈. b = f x ⟹ ∃a ∈ A. b = f a

And the obtained proposition is apparently true. In this way, drule substitutes an existing expression in the premise of the proposition by the conclusion of the rule. Since an original expression in the premise is removed, we cannot use it anymore.

erule: Let us take the following lemma as an example.

```
lemma [ b ∈ B; ∃x ∈ A. b = f x ] ⟹ ∃a ∈ A. f a = b
```

In this lemma, the equation in the conclusion is `f a = b`, whereas that of Example_d is `b = f a`. In this case the equation `b = f x` in `∃x ∈ A. b = f x` and equation `f a = b` in `∃a ∈ A. f a = b` are different in ordering. So, we need more steps to rewrite. At first by apply `(erule bexE)`. Here `bexE` is a rule

```
bexE: "[∃x ∈ A.?P x; ⋀x.[x ∈ ?A; ?P x] ⟹ ?Q] ⟹ ?Q
```

We have the proposition rewritten as

```
⋀x.  [ b ∈ B; x ∈ A; b = f x] ⟹ ∃a ∈ A. f a = b
```

We see the original premise is substituted by the second premise of `bexE`. It is easy to see when we use erule. That is, if there is `∃ x.  P x` or `∃x ∈ A. P x` within a proposition to prove, then we use `erule exE` or `erule bexE`.

frule: Let us treat the next proposition

```
⋀x y.[f ∈ A → B; inj_on f A; g ∈ B → C; inj_on g B; x
∈ A; y ∈ A; g (f x) = g (f y):  f x = f y ] ⟹ x = y
```

To this proposition, we apply a command

```
apply(frule_tac f1 = f and A1 = A and x1 = x and y1 = y
in injective_iff[THEN sym])
```

Here the rule `injective_iff[THEN sym]` is

```
⟦inj_on ?f1 ?A1; ?x1 ∈ ?A1; ?y1 ∈ ?A1⟧ ⟹ (?f1 ?x1 =
?f1 ?y1) = (?x1 = ?y1)
```

As a result, an assumption is added within the premise of the propositon to prove:

```
⋀x y. ⟦f ∈ A ∈ B; inj_on f A; g ∈ B → C; inj_on g B; x
∈ A; y ∈ A; g (f x) = g (f y); f x = f y; (f x = f y) =
(x = y)⟧ ⟹ x = y
```

Thus frule or frule_tac adds conclusion of an applied rule to the premise of the proposition to prove. Our selection method for rules does not work to select rules applied by `frule` or `frule_tac`.

5 Discussion on an automated reasoning

Our automated prover consists of three parts:

> Database of rules, a function to extract proper rules to apply and inference engine

. We think another important function should be added. We explain the function presenting a simple example.
A commutative ring R is called boolean[3] if any element x of R satisfies 2 x = 0, and $x \cdot x = x$. Then we have

> Proposition. If R is boolean, following two hold:
> (1) Any finitely generated ideal is principal.
> (2) Any prime ideal is maximal.

We try Gedanken expreriment to prove these as if we are a machine prover.
(1) Mathematical induction can be used in a prover, we prove the proposition by induction on the cardinality of a generator of the ideal. If it is 1, then

it is trivial. Suppose if the cardinality is n, the ideal is principal, we show the proposition is true is the cardinality is n + 1. Given an ideal $I = (a_1, a_2, \ldots, a_{n+1})$. By virtue of the induction assumption, there is an element a such that $(a_1, a_2, \ldots, a_n) = (a)$. We must show $(a_1, a_2, \ldots, a_n, a_{n+1}) = (a, a_{n+1})$ and $(a, a_{n+1}) = (a + a_{n+1} + a \cdot a_{n+1})$. Now, we suppose in the prover's database, only the definition of boolean is stored. How we can let the prover find the fact $(a, b) = a + b + a \cdot b)$? It seems our prover cannot find out this fact now, because there is no rules concerning to this problem. But since our prover can use LISP, it is not difficult to make a function generating elements obtained by addition and multiplication starting with two elements. In fact starting with two elements a and b, we have 0, a, b, and $a + b$, $a \cdot b$, $a + a \cdot b$, $b + a \cdot b$, $a + b + a \cdot b$ only, generated by binary operations of the ring. We see $a \cdot (a + b + a \cdot b) = a$, and $b \cdot (a + b + a \cdot b) = b$. This implies both a and b are the element of the ideal $(a + b + a \cdot b)$. q.e.d. of (1).
(2) Let P be a prime ideal. In the residue class ring R/P, by the relation $\bar{x}^2 = \bar{x}$, we have $\bar{x} \cdot (\bar{x} - \bar{1}) = \bar{0}$. Since R/P is an integral domain, if $\bar{x}$ is not $\bar{0}$ then $\bar{x} = \bar{1}$. Therefore R/P is a ring with two elements $\{\bar{0}, \bar{1}\}$. It is well known this ring is a field. This implies P is maximal. As a consequence of this experiment, we see
(1) Try to generate elements and then try an element generates every element or not.
(2) Make a residue class ring.
These two are not rules, we call them as mathematical knowledge. So not only three parts already implemented, a prover should have a function to select mathematical knowledge and a function to test along the knowledge.

References

[1] Shunji Kometani, *Set and General Topology (In Japanese)* (Asakura shoten, 2003)

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

[3] M. F. Atiyah and I. G. McDonald, *Introduction to Commutative Algebra* (Addison-Wesley publishing Company, 1969)