Generation of Propositions in Isabelle/HOL

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

1 Intoduction

In Isabelle/HOL, a proof of a mathematical proposition, using backward inferences, proceedes as:

1. take out some theorems, from a database, having conclusions similar to the conclusion of the proposition to prove.
2. apply those chosen theorems one by one to the original proposition and try to rewrite the original proposition. Then choose one of the theorems, rewrite the original proposition and repeat the step 1.
3. if no theorem is found in the DB, we try to generate a lemma which can be applied to the original proposition.

The item 3 is what we are going to discuss in this report. Some theorems are required only for logical calculations. And in a text book, those logical theorems are omitted as trivial, except in some fields related mathematical logic. In Isabelle/HOL automatic provers are equipped and some logical propositions are proved automatically, but if a proposition is complicated, we have to generate a lemma to make the proposition simple.

Moreover, in the set theory there are some simple principle to generate propositions which can be useful later.

We also show a hint, which is given by a mathematician's inspiration, as a proposition used to prove a theorem.

To develop above trials, we take, as an example, the Bernstein's theorem in set theory as a base. We need Set theory, Function theory and some theorems required by logical calculation of Isabelle/HOL.

We make a network of (proved) theorems and axioms, to see the relationships between them. And we note a possibility to give a concise proof by pickking up some propositions in the network.

2 Selection of Theorems from DB

Our prover consists of

1. Isabelle/HOL as a logical rewriting system of a proposition.
2. ProofGeneral, based on emacs, as an interface
3. postgreSQL as a database managing system. Theorems and hints are stored in tables

Between 1 and 3, a proposition is sent by means of data transfer part of the prover written in emacs lisp as:

Isabelle	$\longleftrightarrow$	Emacs Lisp	$\longleftrightarrow$	SQL
proposition	$\longleftrightarrow$	tree	$\longleftrightarrow$	tree in SQL table

A proposition with binary oprator (e.g. $=$, $\wedge$, $\vee$ etc.) is converted to a binary tree, and the other is converted to a linear tree. Here are two examples:

A $\wedge$ B	$\longrightarrow$	(andS (A) (B))
P x	$\longrightarrow$	(P x)

We store trees of propositions already proved in a relational table "propositions" in SQL. And when we have a proposition to prove, we put it in the other table "prop_to_prove". Then we begin to select proper propositions to apply from "propositions".

2.1 Positions of Operators and Variables of a Tree

To compare the conclusion part of the proposition to prove and that of propositions to apply, we compare skeletons of propositions as:

1. make position lists of trees.
2. compare position lists ignoring the difference of variables and the difference of bounded variables.
3. give a similarity point in integer.

where position list is a list of point and its position in the tree expression of the proposition. The skeleton is a list consisting of position and variables which are not operands of the operator variables in the position list. We give an example of position list of the tree expression of a lemma

$$P\ c \Longrightarrow \exists\ \_c\ .\ P\ \_c$$

The tree expression of this proposition is

(LrarS (P c) (exS _x dS P _x))

and position list of the conclusion is

(exS (∼c . _x) (∼c ∼c . dS) (∼c ∼c ∼c . P) (∼c ∼c ∼c ∼c _x))

where (∼c) means the child of the tree, (∼c ∼c) means the child of child of the tree and so on. As position signs, we have ∼l, ∼r, ∼n and P_n, where ∼l the left-child, ∼r right-child, ∼n within parentheses and P_n means in the n-th premise. Hence to reach a variable p with the position (∼c ∼c ∼c . P), we have only to execute a command in emacs-lisp

```
(child (child (child tree)))
```

or a command in SQL

```
select child(child(child(tree)));
```

Instead of the command `child(child(child(tree)))` , we have the command `cutout_subtree(cl, tree)` in SQL, where cl is the list (∼c ∼c ∼c . P). In genera, cl can be any list consists of some elements of a list given as `poss_of_tree(tree)`.

To compare positions of trees, we ignore differences of variables and descendant's position of an operator which appears as cdr of a dot cons.

We present an example to omit a position.

tree	position list
(exS _x dS ?P _x)	(exS (∼c . _x) . . . (∼c ∼c ∼c . ?P) . . .)
(exS _x dS andS (P _x) (Q _x))	(exS (∼c . _x) . . . (∼c ∼c ∼c . andS) . . .)

Here the second line is the tree of the conclusion of a proposition in DB, and the third line is the tree of the conclusion of a proposition to prove. P in the second line is an operator with operand _x, and the subtree (andS (P _x) (Q _x)) is at the same position of P in the proposition to prove. That is the return value of

```
select cutout_subtree('(∼c ∼c ∼c .  ?P)', tree1)
```

, where tree1 is the tree appearing in the third line above.

As noted above, the position of the last bounded variable _x of the tree (exS _x dS ?P _x) should be discarded it is a position of a child of ?P. The reason is that a tree corresponding to ?P can be a complicated tree, and the bounded variable position may be different to that of the child of ?P.

In this example, the counter part of ?P is (lmbS _x1 dS andS (P _x1) (Q _x1)) which is converted as

```
λ _x1 .  P _x1 ∧ Q _x1
```

The tree coresponding to `(?P _x)` is `((lmbS _x1 dS andS (P _x1) (Q _x1)) _x)`. It is easy to see that the position of _x of the former tree and that of the latter is different.

We eleminate those positions not to be compared by using a function.

3 Make A Network of Theorems

In this section, we make a network of axioms and proved theorems. The network makes clear the relationships between theorems.

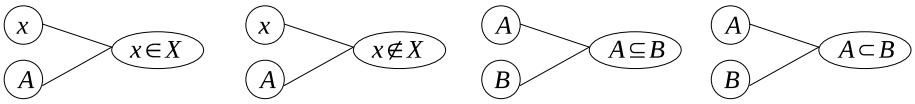
3.1 Elements, Sets and Networks of simple Propositions

notations:
x:: ‘a, x1::’a, x2::’a, ...
y::’b, y1::’b, y2::’b, ...
z::’c, ...
A::’a set, A1::’a set, A2::’a set, ...
B::’b set, B1::’b set, B2::’b set, ... (‘a set is ‘a ⇒ bool)
P::’a ⇒ bool, Q, R,
{x. P x} – a set of elements x with P x true.

Examples:
Elements: 1, 2, 3, ... a, b, c, ... a1, a2, a3, ...
Sets: {}, {1}, {1, 2}, {1, 2, 3}, {a}, {a, b}, {a, b, c}, ...

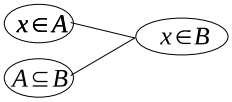
We present some simple propositions expressed in networks.

Membership, subset

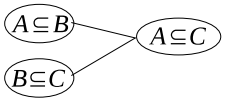


Propositions in a network

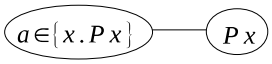
subsetD



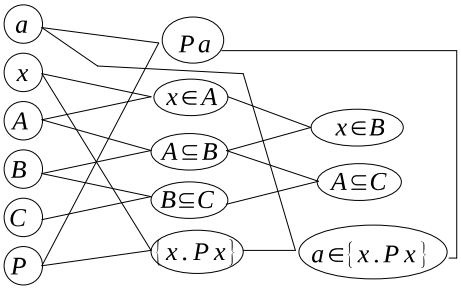
subset_trans



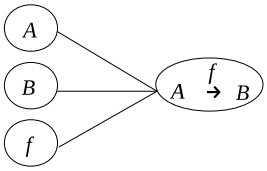
mem_Collect_eq

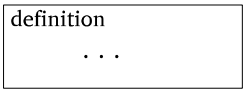


Put above parts together



3.2 Functions

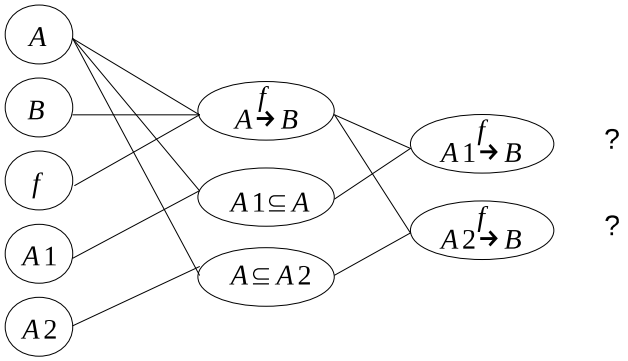




The rectangle implies the definition is not included in the network. However, in Python, we can express the right hand node as `NO1 = {name:function, tree:(inS (f) (rarS (A) (B))), def: (falS inS (x dS inS (x) (B)) (A))}`.

Trial to generate propositions with a simple principle:

1. contraction and extension of the domain

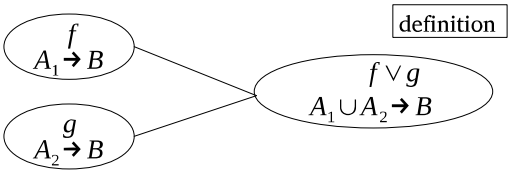


2. contraction and extension of the range. (abbreviated)

Given two functions $f \in A_1 \to B$ and $g \in A_2 \to B$, if $\forall x \in A1 \cap A2.f(x) = g(x)$ then we define

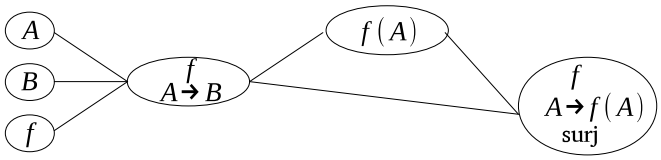
$$f \vee g(x) = \begin{cases} f(x) & \text{if } x \in A_1, \text{ else} \\ g(x) & \text{if } x \in A_2 \end{cases}$$

The network expression of $f \vee g$ is



Note that if A_1 and A_2 are disjoint, f ∨ g is defined without condition.

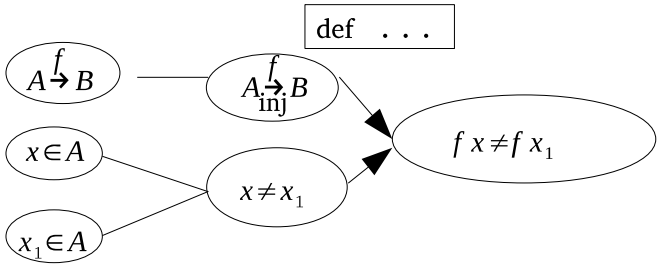
Image of a function:



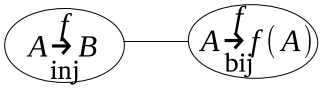
We note that we can try to generate propositions concerning surjection by contraction and extension of a domain and a range.

Suppose the join of functions $f \vee g$ is given and suppose each of the functions are injections, then dividing the domain into three disjoint sets $A_1 - (A_1 \cap A_2)$, $A_2 - (A_1 \cap A_2)$ and $A_1 \cap A_2$, we obtain the following proposition:
$[|\ \forall x \in A_1 \cap A_2.fx = gx;\ \mathtt{inj}_{A_1,B} f;\ \mathtt{inj}_{A_2,B} g;\ \bigwedge x_1, x_2[|x_1 \in A_1 - (A_1 \cap A_2);\ x_2 \in A_2 - (A_1 \cap A_2)|] \Longrightarrow fx_1 \neq gx_2|] \Longrightarrow \mathtt{inj}_{A_1 \cup A_2,B} f \vee g$

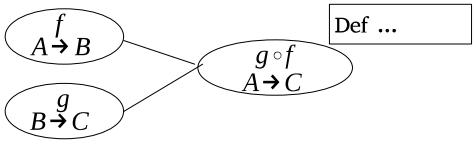
Injection:



Proposition. Inj_bij_to_image:



Composition:

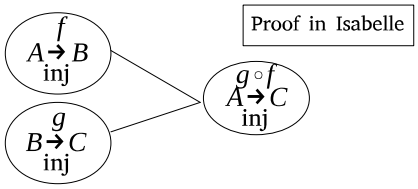


The composition $g \circ f$ above has the following property:

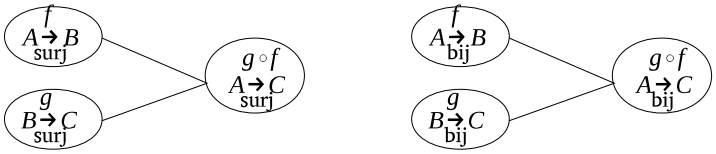
$$\mathrm{im}_{A,C} g \circ f \subseteq \mathrm{im}_{B,C} g$$

.

Proposition comp_inj:



Proposition comp_surj and comp_bij:



We define a relation $\sim_b$ as

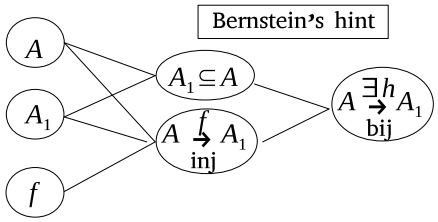
$$A \sim_b B \text{ if and only if there is a bijection from } A \text{ to } B$$

Using Isabelle, we can show this relation is an equivalence relation.

3.3 Bernstein's Hint

Let A_1 be a subset of A, and let f be injective from A into A_1. Make the following subset A_2 of A_1 and make the following join of functions.

$$\begin{array}{lcl} A_2 & = & \{x \in A_1. \exists n \geq 1 \wedge \exists y \in A - A_1 \wedge f^n(y) = x\} \\ f \vee id(x) & = & \begin{cases} f(x) & \text{if } x \in (A - A_1) \cup A_2 \text{ , else} \\ id(x) & \text{if } x \in A_1 - A_2 \end{cases} \end{array}$$

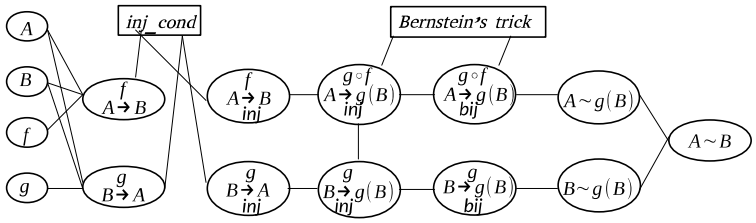


Hence if we have an injection from A into A1, we see A sim_b A1.

3.4 Bernstein's Theorem

Theorem. If we have an injection f from A to B, and an injection g from B to A. Then A $\sim_b$ B. According to a principle "if it is possible to make a composition, make a composition", we make g ∘ f, then we can find `comp_inj` to see that g

∘ f is an injection from A to g(B). and we see B $\sim_b$ g(B). The Bernstein's hint shows A $\sim_b$ g(B). From B $\sim_b$ g(B), we have g(B) $\sim_b$ B. Hence we have A $\sim_b$ B. Since in the network of theorems, the conclusion of Bernstein's theorem is reachable from the elementary objects, we see the theorem is true. Moreover, we see repeated logical calculation can be hidden from the network of theorems, we can give a concise proof to a proposition to prove.



References

[1] Andries P. Engelbrecht (2007) *Calculational Inteligence* John Wiley & Sons Ltd.

[2] Koki Saitoh. (2016) *Deep Learning starting from zero* , O'Reilly Japan Inc.

[3] Bill Lubanovic. (2015) *Introducing Python*, O'Reilly

[4] Tobias Nipkow et al. (2013) *A Proof Assistant for Higher Order Logic*, Springer-Verlag.

[5] Syunji Kametani (2004) *Sets and Topology*, Asakura Shoten