A Resolution Method from Predicate Logic Specification into Executable Code

K. Ono† S. Kawano†† Y. Fukazawa† T. Kadokura†

† School of Science and Engineering
Waseda University
Tokyo, Japan

†† Yamato Laboratory
IBM Japan Ltd.
Tokyo, Japan

Abstract

We present a resolution method from a first order predicate logic formula $F(x, y)$ into a function f as its executable code which computes y from x regarding x as the inputs and y as the outputs. A resolution process of the form $\forall z\ G(x, y, z)$ is argued in particular. This method can be regarded as a theorem proving method for first order predicate logic formulae. From the point of view, the resolution would be the theorem proving of the form "$\forall x\ \exists y\ \forall z\ G(x, y, z)$". The function f is generated by means of the application of the pre-defined rules. These rules can be classified into two groups; transformation rules and resolution rules. The former rules transform a logic formula itself, and the latter rules resolve a function and/or a function definition. In our method, the latter rules are applied first if possible. If no latter rules can be applied, then the former transformation rules are applied. This paper briefly describes some example of the resolution process and verification of its partial correctness.

1 Introduction

A predicate logic based language is promising as a formal specification language [1]. Specification in such language defines characteristics and functions of a given computation system as relations among the elements of the system. Predicate expression is unambiguous, so that specification in such language can be defined unvaguely and exactly.

Most of such languages are based on the first ordered predicate logic(FOPL) [2]. In FOPL, the completeness between formal proof and model theoretical proof is proved, so the formal semantics of such languages can be defined completely. FOPL is powerful enough to represent many computation system.

Although any requirement can be expressed in FOPL exactly and fully, it is difficult for any other developers and users to read it in the design phase and the verification phase. To overcome this disadvantage, many tools to support specification definition are studied. Among them, direct execution of a given specification is promising for specification understanding.

But, it is impossible to transform mechanically any specification in FOPL into an executable code. Therefore the restricted FOPL formulae can be dealt as the object. The proposed methods are Green's resolution method [3], First Order Compiler [4], a resolution method with constructive proof [5] and so on. Green's method resolves functions by theorem proving method. This method proves the restricted form "$\forall x\ \exists y\ F(x, y)$", and resolves the function f which denotes the correspondence between the input x and the output y as the result. In this method, firstly a specification(FOPL formulae) as axioms is transformed into Skolem's standard form, secondly $F(x, y)$ is proved with Robinson's resolution principle [6], and lastly the substituted term for the variable y is obtained as the required function f.

FOPL formula $F(x, y)$ is assumed to be a quantified form "$\forall z\ G(x, y, z)$" or "$\exists z\ G(x, y, z)$". The formula $\forall x\ \exists y\ \exists z\ G(x, y, z)$ is similar as $\forall x\ \exists y' G'(x, y')$, so that we discuss only the form $\forall x\ \exists y\ \forall z\ G(x, y, z)$. Green's method cannot resolve function f from this form, because that cannot obtain the empty clause in spite of selection of correct axiom.

We present a resolution method which applies rules manifoldly. The rules can be classified into two groups; transformation rules and resolution rules. The former rules transform a FOPL formula itself, and the latter rules resolve a functional form and/or a function definition. Firstly, the latter rules are tried to be applied. If no latter rules can be applied, then the former rules

0073-1129-1/92 $3.00

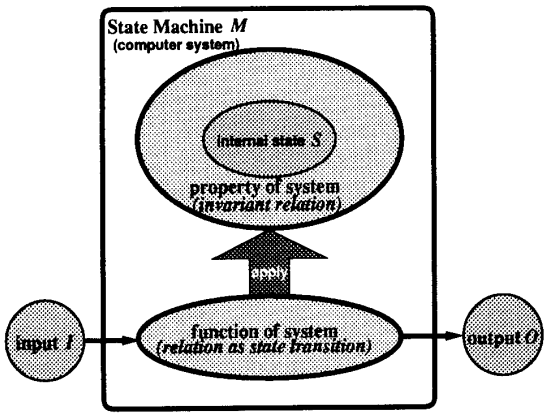

Figure 1: State Machine Model

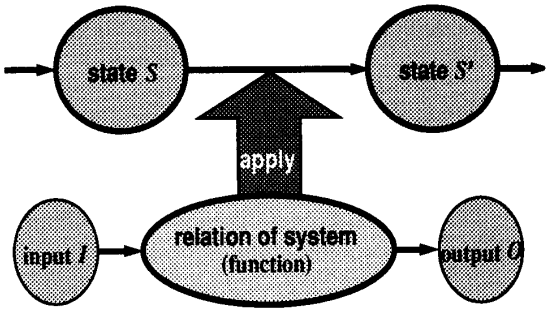

Figure 2: State Transition Model

are applied.

2 Resolution method

2.1 Representation of computation system

We adopted a state machine model as our computation system model (Figure 1). In this model, the property of a computation system is regarded as the invariant relation among elements of the system. And also the function of the computation system is regarded as the relation among the elements and inputs/outputs of the system. An action(computation) of a given system is defined as a state transition (Figure 2); when the internal state of the system is $\mathcal{S}$, the invoked function with inputs $\mathcal{I}$ derives outputs $\mathcal{O}$ and makes the internal state $\mathcal{S}'$.

According to the above definition, when the computation system is specified in FOPL, property and function of the system would be represented by regarding relations as predicates.

2.1.1 Property of system: In order to describe the property, we classified these FOPL formulae for the property into two classes; predicate definition and invariant relation.

Predicate definition This definition names a particular relation as a predicate.

example: a relation that "a person x is *senior to* another person y" will be defined as a formula which represents that "the age of x is *greater than* the age of y". This relation is represented in predicate form "$senior(x, y)$" and "$greater(age(x), age(y))$".
The definition has the following form:

$$\forall x_1, \cdots, x_n \quad [\ p(x_1, \cdots, x_n) \leftrightarrow F_d(x_1, \cdots, x_n)\ ] \tag{1}$$

In (1), the symbol "$\leftrightarrow$" is used as an equivalent symbol, p as a predicate symbol, x_i as a quantified variable, and $F_d(x_1, \cdots, x_n)$ as a FOPL formula including variables $x_1, \cdots, x_n$.

The example is represented in the following.

$$\forall x, y \quad [\ senior(x, y) \leftrightarrow greater(age(x), age(y))\ ] \tag{2}$$

Invariant relation This invariant relation means a constraint for an internal state of a computation system. A state must satisfy a relation at all times.

example: "the number of books in the library stack will be never over 10000," i.e., "the number of books in the library stack is always less than 10000."
The invariant formula has the following form:

$$F_i(s_1, \cdots, s_n) \tag{3}$$

In (3), s_i is used as a state variable which represents the internal state of the computation system at pre-transition.

The example is represented in the following on the assumption that the state variable *books* expresses a set of *book*.

$$size(books) \leq 10000 \tag{4}$$

2.1.2 Function of system: The function of computation system is defined as a relation among its internal states and inputs/outputs of the system. This is represented as a following closed FOPL formula.

example: a function that "a person p check out a book bk from the library" will be defined as a formula which represents that "for all book bk, person p, book's status st, and borrowers list bl, bk must be **returned**, st will be changed to bk *is* **checked out**, and record p *has borrowed* bk *to* bl".

The definition has the following form:

$$\begin{array}{l} \forall s_1, \cdots, s_n, i_1, \cdots, i_o \, \exists {s_1}', \cdots, {s_n}', o_1, \cdots, o_t \\ \quad [\quad F_f(\quad s_1, \cdots, s_n, {s_1}', \cdots, {s_n}', \\ \qquad\qquad i_1, \cdots, i_o, o_1, \cdots, o_t) \quad] \end{array} \tag{5}$$

In (5), ${s_i}'$ is a state variable which represents the internal state of the computation system at post-transition, i_j/o_k are input/output variables for computation system respectively.

The example is represented in the following.

$$\begin{array}{l} \forall bk, p, st, bl \, \exists st', bl' \\ \quad [\quad (\, bk, \textbf{returned} \,) \in st \\ \qquad \wedge \quad st' = st \oplus \{ \, (\, bk, \textbf{checked\_out} \,) \, \} \\ \qquad \wedge \quad bl' = bl \oplus \{ \, (\, bk, p \,) \, \} \quad] \end{array} \tag{6}$$

In (6), "(x, y)" is used as an ordered pair, the symbol "$\oplus$" as relational override.

2.1.3 Action of system: The specification should be represented as the set of equations (1), (3), and (5), and then the action of the system at states $s_1, \cdots, s_n$ is considered to prove the following theorem (7).

$$\begin{array}{l} \forall x_1, \cdots, x_n \quad [\, p(x_1, \cdots, x_n) \leftrightarrow F_d(x_1, \cdots, x_n) \,] \\ \quad \vdash \forall i_1, \cdots, i_o \, \exists {s_1}', \cdots, {s_n}', o_1, \cdots, o_t \\ \quad [\quad F_i(s_1, \cdots, s_n) \\ \qquad \rightarrow \quad F_f(\, s_1, \cdots, s_n, {s_1}', \cdots, {s_n}', \\ \qquad\qquad i_1, \cdots, i_o, o_1, \cdots, o_t \quad) \\ \qquad \wedge \quad F_i({s_1}', \cdots, {s_n}') \quad] \end{array} \tag{7}$$

The evaluated value and state transition by execution of the function f resolved by our method satisfy the consequence of (7) (i.e. following equation (8)).

$$\begin{array}{l} \forall i_1, \cdots, i_o \, \exists {s_1}', \cdots, {s_n}', o_1, \cdots, o_t \\ \quad [\quad F_i(s_1, \cdots, s_n) \\ \qquad \rightarrow \quad F_f(\, s_1, \cdots, s_n, {s_1}', \cdots, {s_n}', \\ \qquad\qquad i_1, \cdots, i_o, o_1, \cdots, o_t \quad) \\ \qquad \wedge \quad F_i({s_1}', \cdots, {s_n}') \quad] \end{array} \tag{8}$$

The purpose of our method is to resolve the following function definition (9) from this consequence.

$$\begin{array}{l} F_d \stackrel{\text{def}}{=} \mu {s_1}', \cdots, {s_n}', o_1, \cdots, o_t \, \lambda s_1, \cdots, s_n, i_1, \cdots, i_o . \\ \quad [\quad F_i(s_1, \cdots, s_n) \\ \qquad \rightarrow \quad F_f(\, s_1, \cdots, s_n, {s_1}', \cdots, {s_n}', \\ \qquad\qquad i_1, \cdots, i_o, o_1, \cdots, o_t \quad) \\ \qquad \wedge \quad F_i({s_1}', \cdots, {s_n}') \quad] \end{array} \tag{9}$$

2.2 Input/output and procedural interpretation of formula

According to the meaning of (8), the inputs and outputs of (8) are defined as the λ bound variables and the μ bound variables of (9), respectively. Therefore, the inputs of (8) are the universally bound variables, which are the inputs $i_1, \cdots, i_o$, and the free variables, which are the internal states $s_1, \cdots, s_n$ of the computation system. Similarly, the outputs of (8) are the existentially bound variables, which are the outputs $o_1, \cdots, o_t$ or the internal states ${s_1}', \cdots, {s_n}'$ of the computation system.

Moreover, the procedural interpretation of (8) gives the computation which substitutes arbitrary values for

the inputs and calculates the output values. This computation is similar to the execution of predicates in (8) as the procedures in an appropriate order. This procedure means to calculate the values of arguments from the others. Some execution orders and some procedural interpretations of a consequence (8) may be considered.

The input and output of a predicate in (8) are defined in the following. A procedural interpretation of each predicate appeared in (8) is procedurally interpreted by adoption of a procedural interpretation of the consequence. The inputs of the predicate are defined as the arguments referred by the procedure, and the outputs of the predicate are defined as the arguments which the procedure calculates as the values.

The inputs and outputs of formula F_d in the predicate definition (1) are equal to the inputs and outputs of the predicate p. The inputs and outputs of the predicate p is determined by procedural interpretation of formulae using p. Therefore, when the specification has some formulae using p, the attributions of arguments in p(the inputs or outputs) may have some varieties.

2.3 Function resolution from FOPL formula

2.3.1 Tree structured standardization: As the pre-process, the consequence in (8) is transformed as follows in order to make it easily applicable for the rules.

A given formula is transformed into a tree structure by regarding a literal as a node, a conjunction as a sequential connection of nodes, and a disjunction as a parallel connection of nodes. It is supposed that each literal is an atomic formula with at most one negation symbol ("¬") on the top. Moreover, a quantified formula is regarded as a atomic formula. We refer to such formula as *Tree Structured Standard Form*, and to such transformation method as *Tree Structured Standardization.*

An example of Tree Structured Standardization is shown in Figure 3. In that, the symbols $A, \ldots, G$ are literals. Figure 4 is a graphically represented result of Figure 3.

When a node(literal) in the Tree Structured Standard Form has some children, the node is called as a *branching point.* The branching points of Figure 4 are G, B *and* C. Moreover, a conjunction of literals from the root to the first branching point of a tree is referred as a *head* of the tree. The head of Figure 4 is $A \wedge D \wedge G$. A tree may have an empty head, $X \vee Y \wedge Z$ for example.

$$(A \wedge B \vee A \wedge C) \wedge D \wedge (E \vee F) \wedge G$$
$$\downarrow$$
$$(A \wedge (B \vee C) \wedge D \wedge (E \vee F) \wedge G$$
$$\downarrow$$
$$A \wedge D \wedge G \wedge (B \vee C) \wedge (E \vee F)$$
$$\downarrow$$
$$A \wedge D \wedge G \wedge (B \wedge (E \vee F) \vee C \wedge (E \vee F))$$

Figure 3: Example of Tree Structured Standardization

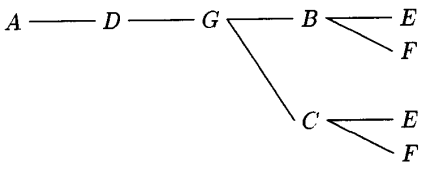

Figure 4: Graphical Representation of Tree Structured Standard Form

The procedural interpretation of Tree Structured Standard Form is that the computation continues the same state until the branching point, and the computation switches the state at that point for each subtree.

2.3.2 Rules: As an executable code, a set of function definitions is generated by finitely applying rules to transform a element of Tree Structure Standard Form into a functional form. The elements of Tree Structure Standard Form are logical connectives, quantifiers, primitives, and user-defined predicates. Rules to generate an executable code are classified into each group for these elements. The rules are applied according to the priority for these groups.

The rules in each group are classified moreover by the objects for application. For example, a group of rules for logical connectives consists of rules for conjunction and rules for disjunction.

Most of the rules are coupled as a pair of a rule to resolve functional form and a rule to transform formula. Resolution rules are given preferences over transformation rules. Figure 5 shows the classification of rules.

Items in a rule are shown in Figure 6.

For example, the following shows a part of the rules for a quantified formula.

> *A quantified formula is transformed by regarding it as an atomic formula. Rules are classified into a group for existentially quantified for-*

- **group of rules for logical connectives**
  - *rules for conjunction*
    - resolution rule
    - transformation rule
  - *rules for disjunction*
    ditto.
- **group of rules for quantifiers**
  - *rule for finite set domain of bound variable*
    - resolution rule
    - transformation rule
  - *rule for infinite set domain of bound variable*
    and so on.
- **groups of rules for primitives**
  - *rule for equality*
  - *rule for set calculation*
    and so on.
- **groups of rules for user-defined predicate**

Figure 5: Classification of Rules

Object
element of Tree Structured Standard Form to apply the rule

Precondition
prior condition when the rule applies

Resolvent
resolved function/function definition, transformed formula

Action
moreover applying other rules, and so on

Figure 6: Items in a Rule

mulae and a group for universally quantified formulae.

rules for existentially quantified formulae

In principle, bound variables for existential quantifier are regarded as the outputs of the formula in the quantified domain. Rules are classified moreover according as the formula can be interpreted procedurally or not, and as the domain of the bound variable is finite or not. There are no transformation rules for existentially quantified formulae.

The following example shows a resolution rule which resolves a procedure to enumerate its elements of finite set and test.

Object
existentially quantified formula:

$$\exists x : S[P(x)]$$

Precondition
the existentially quantified formula cannot be transformed when the bound variable x is regarded as the output of the formula $P(x)$, and the domain S of the bound variable x is finite.

Resolvent

$$\mu z\, \lambda y, S \cdot F(z, y, S)$$

Action
Generate a function definition of F which is

$$\begin{aligned} F \stackrel{\text{def}}{=}\ & \mu z\, \lambda y, S_1 \cdot S_1 = \{\} \rightarrow false : \\ & \lambda e \cdot or(\ F_d(z, y, e), \\ & \qquad F(z, y, S_1 - \{e\})) \\ & (elm(S_1)) \end{aligned}$$

and generate a function definition of F_d which is

$$F_d \stackrel{\text{def}}{=} \mu z\, \lambda y, e \cdot P(z, y, e)$$

The procedure which enumerates an element e from a set S and tests e is generated because the domain of the bound variable x is S. The function $elm(S_1)$ is a primitive which returns an arbitrary element from the set S_1.

rules for universally quantified formulae

The following shows a transformation rule for example which unfolds a quantified formula.

Object
universally quantified formula:

$$\forall x : S \quad [\, P(x) \,]$$

operation definition

$\forall g : GR\ \exists bk : Books$
$[\ at = author(bk) \wedge g = genre(bk)\ ]$

input/output declaration

```
INPUT   GR : P(Geres);
OUTPUT  at : Authors;
```

Figure 7: Example of resolution

Precondition

an output variable of P exists in P, and the domain S of the bound variable x is finite.

Resolvent

$\exists e : S; S_1 : P(P)$
$[\quad sep(S, e, S_1)$
$\wedge\quad P(e)$
$\wedge\quad \forall x : S_1\ [\ P(x)\ ]\ ]$

Action

Nothing

The function $sep(S, e, S_1)$ which is a complemental predicate means that a set S is a union of e as an element of S and S_1 as a set which consists of elements of S except for e.

3 Resolution example

In this section, we will apply our resolution method to a specification of a library stack management system.

Figure 7, which defines an operation, is an examples of the consequence (8). This definition represents a computation; when this operation is invoked with the input GR as a set of genres, an output at as an author's name is computed. The relation between GR and at is that *for all genre g in given GR there exists a work in the library's stack.*

The logical meaning of this operation definition is the following.

$$\forall GR : P(Genres)\ \exists at : Authors$$
$$\forall g : GR\ \exists bk : Books$$
$$[\ at = author(bk) \wedge g = genre(bk)\ ] \quad (10)$$

In (10), $x : S$ is a declaration that the domain of a variable x is a set S. Further, $P(S)$ represents the powerset of a set S. Therefore $x : P(S)$ declares that the value of a variable x is a subset of the set S. *Genres* is a set of genres on all publications, *Authors* is a set of all authors, and *Books* is a set of books in the library's stack. *author* is a function which relates a book to its author, *genre* is a function which relates a book to its genre.

It is obvious that the formula has the form $\forall x\ \exists y\ \forall z\ G(x, y, z)$. When this formula is the consequence (8), the formula contained in this consequence, whose form is $\forall z\ G(x, y, z)$, is as follows.

$$\forall g : GR\ \exists bk : Books$$
$$[\ at = author(bk) \wedge g = genre(bk)\ ] \quad (11)$$

Our method attempts to apply the rules which generate a functional form. The required rule is selected from the group for a universally quantified formula. According to the inputs/outputs of (11), it becomes known that the formula

$$\exists bk : Books$$
$$[\ at = author(bk) \wedge g = genre(bk)\ ] \quad (12)$$

has an output variable bk. Furthermore, according to the definition of the set S, it becomes known that S is a finite set. These lead to apply the transformation rule to *unfold a quantified formula*, so that (11) is transformed into the following.

$$\exists g1 : GR; GR0 : P(GR)$$
$$[\quad sep(GR, g1, GR0)$$
$$\wedge\quad \exists bk : Books$$
$$[\quad at = author(bk)$$
$$\wedge\quad g1 = genre(bk)\ ]$$
$$\wedge\quad \forall g : GR0\ \exists bk : Books$$
$$[\quad at = author(bk)$$
$$\wedge\quad g = genre(bk)\ ]\ ] \quad (13)$$

Next, a resolution rule is selected from the group for *sequential execution of conjunction.* Since "$sep(GR, g1, GR0)$" is a complement predicate which is introduced by the above application of the transformation rule, the rule for *primitive/complement predicate* is selected and applied. This generates the following function definition.

$$f1 \stackrel{\text{def}}{=} \lambda GR, Books \cdot$$
$$\lambda g1 \cdot f2(g1, setdiff(GR, g1), Books)$$
$$(elm(GR)) \quad (14)$$

The function $f2$ is defined in (16). The resolution rules are applied moreover. In the quantification of the remaining formula

$$\exists bk : Books \quad [\, at = author(bk) \wedge g1 = genre(bk) \,] \quad (15)$$

the predicate "$at = author(bk)$" cannot be transformed/resolved when the bound variable bk is regarded as the output of this predicate. Furthermore, as $Books$ is a finite set, the resolution rule which generates an iteration by recursion is applied to (15). This application resolves the following function definitions.

$$\begin{aligned}
f2 &\stackrel{\text{def}}{=} \lambda g1, GR0, Books \cdot f2_1(g1, GR0, Books) \\
f2_1 &\stackrel{\text{def}}{=} \lambda g1, GR0, Books \cdot Books = \{\} \rightarrow false : \\
&\quad \mu at\ \lambda bk \cdot or(and(f2_2(g1, bk, at), \\
&\qquad f3(GR0, at, g1)), \\
&\qquad f2_1(g1, GR0, Books - \{bk\})) \\
&\quad (elm(Books)) \\
f2_2 &\stackrel{\text{def}}{=} \lambda g1, bk, at \cdot and(g1 = genre(bk), \\
&\qquad at = author(bk)) \qquad (16)
\end{aligned}$$

The resolution rule for *sequential execution of conjunction* is applied next. From (13) the remaining formula

$$\forall g : GR0\ \exists bk : Books \quad [\, at = author(bk) \wedge g = genre(bk) \,] \quad (17)$$

has no output in the domain. Moreover, because GR0 is a finite set, the resolution rule which *generates an enumeration and testing by recursion* is applied. At the same time, the rule which *generates an iteration and finding by recursion* is applied to the formula in the domain,

$$\exists bk : Books \quad [\, at = author(bk) \wedge g = genre(bk) \,] \quad (18)$$

These applications lead to the following function definitions.

$$\begin{aligned}
f3 &\stackrel{\text{def}}{=} \lambda GR0, at \cdot f3_1(GR0, at) \\
f3_1 &\stackrel{\text{def}}{=} \lambda GR0, at \cdot GR0 = \{\} \rightarrow true : \\
&\quad \lambda g \cdot and(f4(Books, at, g), \\
&\qquad f3_1(GR0 - \{g\}, at)) \\
&\quad (elm(GR0)) \\
f4 &\stackrel{\text{def}}{=} \lambda Books, at, g \cdot f4_1(Books, at, g) \\
f4_1 &\stackrel{\text{def}}{=} \lambda Books, at, g \cdot Books = \{\} \rightarrow false : \\
&\quad \lambda bk \cdot or(f4_2(g, bk, at), \\
&\qquad f4_1(Books - \{bk\}, at, g)) \\
&\quad (elm(Books)) \\
f4_2 &\stackrel{\text{def}}{=} \lambda g, bk, at \cdot and(g = genre(bk), \\
&\qquad at = author(bk)) \qquad (19)
\end{aligned}$$

These applications of rules resolve the form $\forall x\ \exists y\ \forall z\ G(x, y, z)$ into the set of function definitions. The function definitions are executable by suitable function evaluator.

4 Evaluation

4.1 Termination of resolution

According to our definition of specification in section 2.1, it is obvious that a specification consists of a finite set of FOPL formulae, and each FOPL formula consists of a finite set of predicates, logical connectives, and quantifiers.

Tree Structured Standardization always terminates because it only transforms elements of logic formula finitely. Furthermore, the applicable rules for one element of formula are finite. The application of resolution rules terminates finitely because one of these rule for one element resolves a functional form and/or function definition on each application. It is possible that a transformation rule is applied to a element iteratively.

The groups of such rules are the following.

- reordering of conjunction/disjunction
- unfolding of universally quantified formulae

According to the definition of the former, it is obvious that the application of these rules terminates in n-times(n is a number of literals in the conjunction/disjunction). According to the precondition and

the priority of the latter, it becomes known that the application of its rule terminates. Termination of our method was shown.

4.2 Correctness of resolution

Partial correctness of the resolution is verified below. "*The resolution is partially correct.*", i.e. "*a resolved executable code holds partial correctness for the specification.*" means that "*when the output of the executable code can be obtained for all possible inputs, these inputs and outputs satisfy the logic formula given as the specification.*" The followings investigate our method's partial correctness as a relation between a specification which is a set of FOPL formulae and an executable code which is a set of function definitions.

The model $\mathcal{M}$ of S as a set of FOPL formulae determines the interpretation on the Herbrand universe $H(S)$ for a predicate which appears in S. In the set F of function definitions, the function f which calculates a FOPL formula $\forall x\ \exists y\ [L(x,y)]$ is

$$f \stackrel{\text{def}}{=} \mu y\ \lambda x \cdot L(x,y) \qquad (f \in F) \qquad (20)$$

The meaning that this function f holds partial correctness for the specification S is

$$\mathcal{M}(S) \models L(a,b) \quad if \quad \mathcal{M}(F) \models b = f(a) \qquad (21)$$

holds in arbitrary Herbrand atomic clause L(a,b) including L.

Because the Tree Structured Standardization is the equivalent transformation of FOPL formula $L(x,y)$, the Tree Structured Standard Form holds total correctness. Therefore we discuss correctness of the rules.

The most important resolution rules are those for quantified formula. The following shows those rules.

1. a rule which generates a searching function by considering that a bound variable in existentially quantified formula is a free variable
2. a rule which generates a recursion for existentially quantified formula as an iteration for finding out a required element
3. a rule for existentially quantified formula whose domain of a bound variable is an infinite set
4. a rule which generates a recursion for universally quantified formula as an enumeration for testing all elements
5. a rule for universally quantified formula whose domain of a bound variable is an infinite set

The rules 1,2 and 4 hold correctness according to the logical meaning of quantifiers. However, functions which are generated by the rules 3 and 5 cannot assure the termination to execute itself. Consequently, the resolution is only partially correct because the solution satisfies $L(x,y)$ when the execution terminates, but it is incomplete for arbitrary inputs of $L(x,y)$.

5 Conclusion

We have discussed the outline of our resolution method from the FOPL based specification into function definitions as an executable code. For a detailed evaluation, the specification of library's stack management system has been described, which consists of 11 operation definitions/52 predicates. We have applied this method, so that the whole executable code, which consists of 25 function definitions, can be generated.

Addition of rules extends the resolvable region, however, it may make a failure in correctness of the resolution. Therefore, rules must be carefully added.

References

[1] B. Cohen, W. T. Harwood and M. I. Jackson, "*The Specification of Complex System*", Addison-Wesley Publishing Company, 1986.

[2] M. Fitting, "*First-Order Logic and Automated Theorem Proving*", Springer Verlag, 1990.

[3] C. Green, "*Application of Theorem Proving to Problem Solving*", Proc. 1st IJCAI, pp.219–239, May 1969.

[4] T. Sato and H. Tamaki, "*First Order Compiler*", Computer Software, vol.5, no.2, pp.69–80, 1988.

[5] S. Goto, "*Program Synthesis from Natural Deduction Proofs*", Proc. 6th IJCAI, pp.339–341, 1979.

[6] J. A. Robinson, "*A Machine-Oriented Logic Based on Resolution Principle*", J. of the Assoc. for Comput. Machinery, vol.12, no.1, pp.23–41, January 1965.