ICOT Technical Memorandum: TM-0154

TM-0154

A Proof Constractor for Intensional Logic ··· with S5 decision procedure ···

by
H. Sawamura (Fujitsu Ltd.)

January, 1986

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Proof Constructor for Intensional Logic
— with S5 decision procedure —

By

Hajime Sawamura

International Institute for Advanced Study of Social Information Science (IIAS-SIS), Fujitsu Ltd., Numazu, Shizuoka 410-03 JAPAN

Abstract

Much work has been devoted to computer-aided reasoning systems for extensional logic. This paper reports a proof constructor (PCIL) for the intensional logic (IL) devised by R. Montague. It supports interactive construction of derivations of IL, as well as proof checking capability. It also incorporates a decision procedure for S5 modal logic, which helps us shorten lengthy and tedious proof steps.

Prolog is employed as a logical system description language, and the language and deductive system of IL are described in the relational framework of Prolog. The relational representation of the axiom system of IL has allowed us to easily implement PCIL with the help of Prolog's nondeterministic computational mechanism. Some utilizations of PCIL are demonstrated through practical examples, including proofs from PTQ and combinatory logic.

1. Introduction

The main roles of logics in computer science and artificial intelligence are to provide languages with high expressive power and reasoning (computational) mechanism. In recent years, nonclassical logics have been imported into a great many areas of computer science and artificial intelligence [1].

Nonclassical logics generally concern various aspects of meaning of objects, other than those treated by classical logics. For example, as to truthhood of propositions, they deal with time or situation-dependent truth, vague truth, constructive truth, etc.

The intensional logic [2,3] is a kind of modal logic built on the theory of types, which was originally formalized for incorporating two aspects of a meaning, intension and extension, into a formal system, and for logical analysis of linguistic expressions. Due to the potential expressive power that was inherited from modal logic and the theory of types, the intensional logic has provided the semantical basis for various areas in computer science. In fact, the relevance of the intensional logic to several applications can be seen in programming language semantics and verification [4-7], database semantics [8] etc., as well as in computational linguistics [9].

In spite of this significance of the intensional logic, there has been little work in the area of checking or automating the proofs in the intensional logic, although much work has been done on a proof checker for extensional logics [10-13]. This paper is devoted to the construction of a reasoning system for the intensional logic. It also incorporates a decision procedure for S5 modal logic [14], which helps us shorten lengthy and tedious proof steps. In this paper, we exploit Prolog [15] both as a logical system description language and as an implementation language. The declarative and procedural semantics of Prolog allow us to deal with various concepts of logics in a single uniform framework, and hence easily realize it on a computer. Also, recursive definitions often appearing in logic can be corresponded to those in Prolog, in a straightforward manner.

The paper is organized as follows. Section 2 describes a slightly modified

version of the intensional logic formalized by Gallin [2], in the relational framework of Prolog. This section also includes an outline of our decision procedure for S5 modal logic which has been incorporated into our Proof Constructor for Intensional Logic (PCIL). PCIL is briefly described in Section 3, together with three proof checking examples. The final section includes concluding remarks.

2. Intensional logic in Prolog

Gallin [2] introduced an axiom system for a tenseless version of Montague's intensional logic [3]. It corresponds very closely to the axiomatization for the theory of propositional types given in Henkin [16], as simplified in Andrews [17]. Henceforce, we will call Gallin's intensional logic IL. The language and the style of the axiom system of IL are equational. Therefore, certain parts of equational reasoning of IL may be effectively achieved by employing a term rewriting system [e.g. 21]. It is also associated with possible worlds (indexical) semantics.

In this section, we outline how intensional logic can be specified in the relational framework of Prolog. Its detailed specification is given in [18, 19]. We assume that the readers have some familiarity with the syntax and semantics of IL and Prolog.

The variable symbols and constant symbols, together with their syntactical variables, will be specified when necessary. Besides, we use the special symbols : and, or, imp, not (logical connectives), all, some (quantifiers), int (intension), ext (extension), = (equality), lambda (function abstraction), # (fuction application) (the operator precedences and operator types of these symbols are defined using Prolog's operator declaration.) In what follows, Prolog variables play the role of metametavariables ranging over objects and metaobjects of IL.

The syntax of types is defined as follows:

```
type(e).
type(t).
type((X,Y)) :- type(X),type(Y).
```

```
type((s,X)) :- type(X).
type(T) :- type_var(T).
```

where the last clause allows us to prove theorem schemas (or polymorphic theorems) together with the metasymbols ranging over terms. (see [20] for the usefulness of type polymorphism in programming.) As type variables, we use the symbols of the form of ti (i>0).

A term A with type T is written as A:T. A few steps of the inductive definition which specifies the set term(A,T) of terms A:T in PCIL is as follows :

```
term(A # B,S) :- term(A,(T,S)),term(B,T).
term(all(X:S,A),t) :- variable(X:S),term(A,t).
term(int(A),(s,T)) :- term(A,T)., etc.
```

A formula of IL is a term of type t, that is,

```
formula(F) :- term(F,t).
```

In PCIL, simple type inference is accomplished by unification, for consistently manipulating terms of IL. This is easily done for IL, since a basic type is assigned to every primitive symbols. For example,

```
type_inference(ext(A),ext(B:(s,T)) :- type_inference(A,T).
```

The axioms of IL are specified by the following six clauses :

```
axiom(ax1,*g:(t,t) # *true:t and *g(t,t) # *false:t = all(*x:t,*g:(t,t) # *x:t)).
axiom(ax2,*x:t1 = *y:t1 imply *f:(t1,t) # *x:t1 = *f:(t1,t) # *y:t1).
axiom(ax3,all(*x:t1,*f:(t1,t2) # *x:t1 = *g:(t1,t2) # *x:t1) = (*f:(t1,t2) = *g:(t1,t2))).
axiom(ax4,lambda(x:t1,a:t2) # b:t1 = c:t2) :- subst(a:t2,x:t1,b:t1,c:t2).
axiom(ax5,nec(ext(*f:(s,t1)) = ext(*g:(s,t1))) = (*f:(s,t1) = *g:(s,t1))).
axiom(ax6,ext(int(a:t1)) = a:t1).
```

where symbols with * represent object symbols, symbols without * represent meta symbols, and subst(A,X,T,B) is assumed to be one of the primitive symbol-manipulating predicates, which means that B is a term obtained by substituting T for variables X's occurring in A. Note that these axioms are identical with those of

IL, except the fourth axiom. The predicate subst is recursively defined so as to satisfy the condition for the beta-conversion rule of IL. For example,

```
subst(int(A:S):(s,S),X:T,B:T,int(C:S):(s,S)) :- mc(B:T),subst(A:S,X:T,B:T,C:S).
```

where mc(A) means that A is a modally closed term whose meaning is independent of possible worlds.

The axiom ax4 is actually defined in IL as

$(\text{lambda } x_a\ A_b(x_a))B_a = A_b(B_a)$

where $A_b(x_a)$ denotes a term with type b involving the variable x_a with type a, $A_b(B_a)$ comes from $A_b(x_a)$ by replacing all free occurrences of x_a by the term B_a, B_a is free for x_a in $A_b(x_a)$, and either no free occurrence of x_a in $A_b(x_a)$ lies within the scope of int, or else B_a is modally closed.

The reason we have employed the above definition for ax4 is due to the fact that it is difficult to define, directly as a pattern for pattern matching, such a metaterm as $A_b(x_a)$. In this paper, the metaterm $A_b(x_a)$ is equivalently expressed in terms of lambda abstraction and function application as

```
lambda(x:a,A:b) # x:a
```

So, the metatheorem in IL, for example,

$\vdash \text{all } x_a\ A(x_a) \Rightarrow \vdash A(B_a)$, with the same proviso as the axiom ax4,

is represented in PCIL as

```
|- all(x:a, lambda(x:a, A:b) # x:a)  =>  |- lambda(x:a, A:b) # B:a
```

Therefore, our definition has the advantage that the above conditions for substitution can be implicitly absorbed into such a beta-conversion.

An inference rule can be given a procedural meaning like Prolog, when it is viewed as a relation between antecedents and a conclusion. For example, the universal instantiation rule,

$$\frac{\text{all } x\ p(x)}{p(t)}, \text{ where } t \text{ is free for } x \text{ in } p(x),$$

can be literally described as

```
universal_instatiation_rule(all(X,P),X,T,Q) :- free_for(T,X,P),subst(P,X,T,Q).
```

In fact, the single inference rule in PCIL

$$\frac{A{:}t = B{:}t \quad C{:}t}{D{:}t}$$

where D:t is a formula obtained by replacing one occurrence of A:t (not immediately preceded by lambda) in C:t by the term B:t, is procedurally defined as follows:

```
inference_rule(A:T =B:t,C:t,D:t) :- replace(A:T =B:t,C:t,D:t).
```

where replace is assumed to be another primitive symbol-manipulating operation and is defined by the structural induction on formulas. For example, a few steps of induction are as follows:

```
replace(A:T = B:T,(C:(S,R) # D:S):R,(C:(S,R) # E:S):R) :-
        replace(A:T = B:T,D:S,E:S).
replace(A:T = B:T,(C:(S,R) # D:S):R,(E:(S,R) # D:S):R) :-
        replace(A:T = B:T,C:(S,R),E:(S,R)).
```

Such a definition is best suited to the nondeterministic replacement by the inference rule. If needed, the nondeterministic replacement can be made deterministic by means of a Prolog's cut symbol [22]. Derived rules can be defined in the same manner. For example, the rule of symmetricity becomes

```
derived_rule(A:t = B:t,B:t = A:t).
```

A proof is a sequence of formulas associated with justifications, which is described in PCIL in the following form:

```
theorem | derived_rule <name>. |- <formula>.
    proof.
      1 |- <formula>  by <justification>.
      2 |- <formula>  by <justification>.
          ---------
      n |- <formula>  by <justification>.
    end.
```

where <justifiction> represents one of the followings : as (assumption), ax (axiom), ir

(inference rule), dr (derived rule), th (theorem), def (definition), rw (rewrite), tppml (theorem prover for S5 modal logic), etc.

"rw" denotes a distinguished derived rule such that the right hand side of an equation A = B is rewritten by using an appropriate equation, either of whose sides matches B.

In IL, the modal operator nec can be defined as

nec(A) = [int(A) = int(true)].

Thus, nec is an S5 modal operator. In PCIL, the validity of an S5 propositional modal formula is checked by calling tppml, the decision procedure for S5 modal logic (actually, we can prove a formula in either of the system T, S4 or S5.) The underlying system for the decision procedure is called Sequence Calculus [14], which is derived from Gentzen-style formal system for modal logic. The construction of Sequence Calculus depends on the fact that every formula of degree higher than first is reducible in S5 to a first-degree formula [23] and for such a reduced formula cut elimination theorem holds in S5 [24]. Then, it consists of the axiom schema **S**,A,**T**,not(A),**R** and the following inference schemata :

$$\text{(Thinning)}\ \frac{\mathbf{S}}{\mathbf{S},A} \qquad \text{(notnot)}\ \frac{\mathbf{S},A}{\mathbf{S},\text{not}(\text{not}(A))}$$

$$\text{(and)}\ \frac{\mathbf{S},A \quad \mathbf{S},B}{\mathbf{S},A \text{ and } B} \qquad \text{(notand)}\ \frac{\mathbf{S},\text{not}(A),\text{not}(B)}{\mathbf{S},\text{not}(A \text{ and } B)}$$

$$\text{(nec)}\ \frac{\text{not}(\text{nec}(\mathbf{S})),\text{nec}(\mathbf{T}),A}{\text{not}(\text{nec}(\mathbf{S})),\text{nec}(\mathbf{T}),\text{nec}(A)} \qquad \text{(notnec)}\ \frac{\mathbf{S},\text{not}(A)}{\mathbf{S},\text{not}(\text{nec}(A))}$$

where a boldface letter denotes a sequence of formulas, and a sequence of formulas is identified with a set of formulas. It should be noted that our decision procedure is derived from proof-theoretic considerations, in contrast with other methods based on model-theoretic concepts [25-27].

Here is a proof example :

```
theorem s5. |- pos(nec(p:t)) imp nec(pos(p:t)).
  proof.
    1 |- pos(nec(p:t)) imp nec(pos(p:t))  by tppml.
  end.
```

Actually, the corresponding first-degree formula nec(p:t) imp pos(p:t) turns out to be proved, and then the proof tree is constructed as follows :

```
   not(p:t),p:t
------------------------------  (notnec)
   not(nec(p:t)), p:t
-----------------------------------  (notnot)
   not(nec(p:t)), not(not(p:t))
-----------------------------------  (notnec)
   not(nec(p:t)), not(nec(not(p:t)))
```

3. A proof constructor for intensional logic (PCIL)

3.1 Outline of the system PCIL

Based on the description of IL in Section 2, PCIL has been implemented on Vax11/780 in CProlog. The structure of PCIL system is completely modularized as depicted in Figure 1. It shows that PCIL is a general purpose proof constructor in the sense that the boxes defining the logical system of IL may be replaced by those of other logical systems. See (3) below for such a use of PCIL.

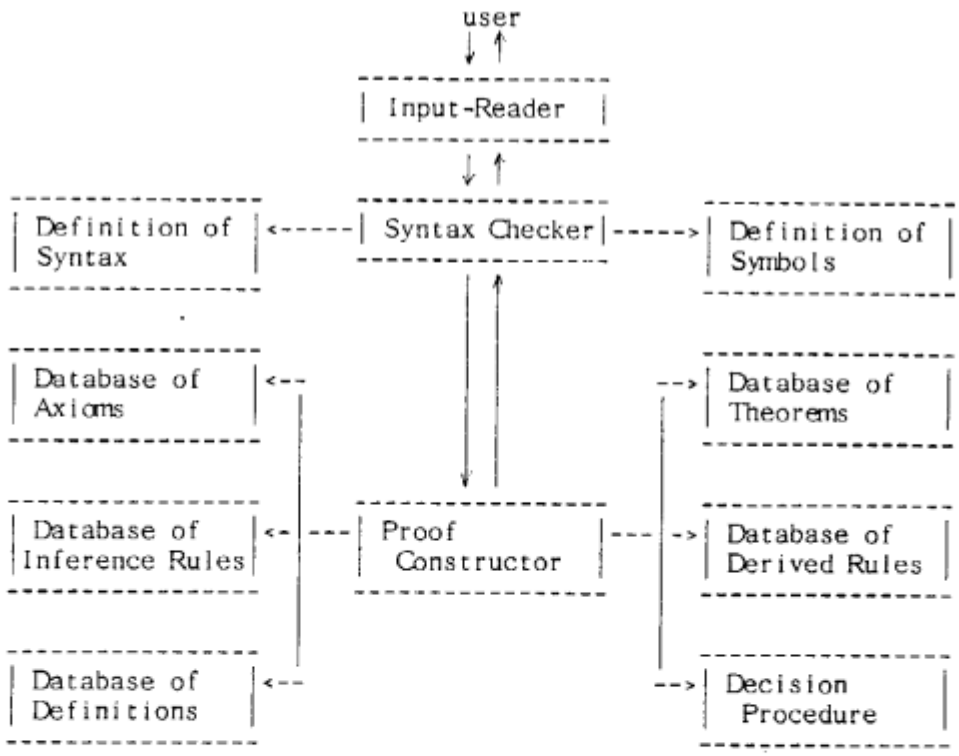

Figure 1 : Structure of PCIL

The Input-Reader reads and analyses commands or proof lines from a user. A user can either construct a proof line by line or check a proof at a time.

3.2 Proof-checking examples

Here, we show three completed proof examples. In the following, unknown names appearing in the justifications, such as th7, dr1, etc., refer to the theorems or derived rules which has already been proved elsewhere. Their referents, however, could be inferred from the context.

(1) Universal generalization theorem schema in IL

```
derived_rule dr3. |- a:t => |- all(x:t1, a:t).
  proof.
    1 |- a:t   by as.
    2 |- (a:t = *true:t) = a:t   by (th,th7).
    3 |- a:t = (a:t = *true:t)   by (dr,dr1,2).
    4 |- a:t = *true:t   by (ir,3,1).
    5 |- lambda(x:t1,a:t) = lambda(x:t1,a:t)   by (th,th1).
    6 |- lambda(x:t1,a:t) = lambda(x:t1,*true:t)   by (ir,4,5).
    7 |- all(x:t1,a:t)   by def.
  end.
```

(2) Reduction of a natural language sentence

Let us see the reduction of a natural language sentence in PCIL. In Montague's PTQ, the following English sentence

John believes that a fish walks.

is translated into an IL expression (omitting types)

lambda P some x [fish(x) & P{x}](^lambda y believe(j,^walk(y)))

This is further reduced to

some x [fish(x) & believe(j,^walk(x))].

The correctness of such an reduction can be checked step by step as follows.

```
Theorem ptq. |- some(*x:e,(*fish:(e,t) # *x:e) and *believe:((s,t),(e,t)) #
                          int(*walk:(e,t) # *x:e) # *j:e).
  proof.
   1 |- lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
          ext(*p:(s,(e,t))) # *x:e))) #
            int(lambda(*y:e,*believe:((s,t),(e,t)) #
              int(*walk:(e,t) # *y:e) # *j:e))   by as.

   2 |- lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
          ext(*p:(s,(e,t))) # *x:e))) #
            int(lambda(*y:e,*believe:((s,t),(e,t)) #
              int(*walk:(e,t) # *y:e) # *j:e)) =
        some(*x:e,(*fish:(e,t) # *x:e and
          ext(int(lambda(*y:e,*believe:((s,e),(e,t)) #
            int(*walk:(e,t) # *y:e) # *j:e))) # *x:e))   by (ax,ax4).

   3 |- ext(int(lambda(*y:e,*believe:((s,e),(e,t)) #
          int(*walk:(e,t) # *y:e) # *j:e)) =
```

```
        lambda(*y:e,*believe:((s,e),(e,t)) #int(*walk:(e,t) # *y:e) # *j:e)
        by (ax,ax6).

    4 |- lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
         ext(*p:(s,(e,t))) # *x:e))) #
           int(lambda(*y:e,*believe:((s,t),(e,t)) #
             int(*walk:(e,t) # *y:e) # *j:e)) =
       some(*x:e,(*fish:(e,t) # *x:e and
         lambda(*y:e,*believe:((s,e),(e,t)) #
           int(*walk:(e,t) # *y:e) # *j:e) # *x:e))    by (ir,3,2).

    5 |- lambda(*y:e,*believe:((s,e),(e,t)) #
           int(*walk:(e,t) # *y:e) # *j:e) # *x:e =
       *believe:((s,e),(e,t)) # int(*walk:(e,t) # *x:e) # *j:e    by (ax,ax4).

    6 |- lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
         ext(*p:(s,(e,t))) # *x:e))) #
           int(lambda(*y:e,*believe:((s,t),(e,t)) #
             int(*walk:(e,t) # *y:e) # *j:e)) =
       some(*x:e,(*fish:(e,t) # *x:e and
         *believe:((s,e),(e,t)) # int(*walk:(e,t) # *x:e) # *j:e))    by (ir,5,4).

    7 |- some(*x:e,(*fish:(e,t) # *x:e) and
         *believe:((s,e),(e,t)) # int(*walk:(e,t) # *x:e) # *j:e)    by (ir,6,1).
  end.
```

(3) Reduction of a typed combinatory term [28]

In order to show the generality of PCIL as a proof constructor, we will construct a proof of a theorem, SKK = I in typed combinatory logic, simply with the following additional axioms :

```
(1) axiom(ax7,i:(t1,t1) # x:t1 = x:t1)

(2) axiom(ax8,k:(t1,((t2,t3),t1)) # x:t1 # y:(t2,t3) = x:t1)

(3) axiom(ax9,s:((t1,((t2,t4),t3)),((t1,(t2,t4)),(t1,t3))) #

      f:(t1,((t2,t4),t3)) # g:(t1,(t2,t4)) # x:t1 = f:(t1,((t2,t4),t3))

        # x:t1 # (g:(t1,(t2,t4)) # x:t1))

theorem c1. |- s:((t1,((t2,t1),t1)),((t1,(t2,t1)),(t1,t1))) #
                 k:(t1,((t2,t1),t1)) # k:(t1,(t2,t1)) = i:(t1,t1)
  proof.
   1 |- x:t1 = x:t1   by (th,th1).
   2 |- x:t1 = i:(t1,t1) # x:t1   by (rw,1,ax7).
   3 |- i:(t1,t1) # x:t1 = x:t1   by (dr,dr1,2).
   4 |- i:(t1,t1) # x:t1 = (k:(t1,((t2,t1),t1)) # x:t1) # (k:(t1,(t2,t1))
          # x:t1)   by (rw,3,ax8).
   5 |- i:(t1,t1) # x:t1 =
        s:((t1,((t2,t1),t1)),((t1,(t2,t1)),(t1,t1))) #
          k:(t1,((t2,t1),t1)) # k:(t1,(t2,t1)) # x:t1   by (rw,4,ax9).
   6 |- s:((t1,((t2,t1),t1)),((t1,(t2,t1)),(t1,t1))) #
          k:(t1,((t2,t1),t1)) # k:(t1,(t2,t1)) # x:t1 = i:(t1,t1) # x:t1   by (dr,dr1,5).
   7 |- all(x:t1,s:((t1,((t2,t1),t1)),((t1,(t2,t1)),(t1,t1))) #
        k:(t1,((t2,t1),t1)) # k:(t1,(t2,t1)) # x:t1 = i:(t1,t1) # x:t1))   by (dr,dr3,6).
```

```
8 |- all(x:t1,s:((t1,((t2,t1),t1)),((t1,(t2,t1)),(t1,t1))) #
     k:(t1,((t2,t1),t1)) # k:(t1,(t2,t1)) # x:t1 = i:(t1,t1) # x:t1)
     = (s:((t1,((t2,t1),t1)),((t1,(t2,t1)),(t1,t1))) #
          k:(t1,((t2,t1),t1)) # k:(t1,(t2,t1)) = i:(t1,t1))   by (ax,ax3).
9 |- s:((t1,((t2,t1),t1)),((t1,(t2,t1)),(t1,t1))) #
          k:(t1,((t2,t1),t1)) # k:(t1,(t2,t1)) = i:(t1,t1)   by (ir,8,7).
end.
```

4. Concluding remarks

We have presented a proof constructor for the intensional logic and showed its usefulness and generality through three proof examples. PCIL is expected to contribute to various areas requiring intensional analysis of objects, from an aspect of reasoning mechanism. On the other hand, the underlying logics of FOL, LCF, PL/CV, EKL, etc., are all extensional, and they provide us with the distinctive apparatuses for formal reasoning, symbolic computation, program verification, etc.

Constructing proofs by analogy to existing proofs, extracting proof templates by induction, powerful proof/formula editor for computer-aided reasoning, etc., are left as future research themes.

It should be observed that the method in this paper, which is to describe the intensional logic, can be applied to other various logics as well. In fact, we have convinced that Prolog is not only natural but also expressive as a logical system description language. Furthermore, it suggests a realization of a metasystem which could allow one to reason according to his own logic, once he described it in terms of a logical system description language such as Prolog.

Acknowledgements

The author would like to acknowledge the continuing guidance and encouragement of Dr. T. Kitagawa, the president of his institute and Dr. H. Enomoto, the director of his institute. The author would also like to thank Dr. M. Toda, Mr. T. Minami for their helpful discussions and comments on computer-aided reasoning systems including this work. This work is part of a major R & D project of the Fifth Generation Computer, conducted under a program set up by the MITI.

References

[1] Turner, R. : Logics for artificial intelligence, Ellis Horwood Limited, 1984.

[2] Gallin, D. : Intensional and higher-order modal logic with applications to Montague semantics, North-Holland mathematics studies 19, 1975.

[3] Tomason, R. H. (ed.) : Formal philosophy - selected papers of R. Montague,

Yale Univ. Press, 1977.

[4] Janssen, T. M. V. and Boas, P. van Emde : On the proper treatment of referencing, dereferencing and assignment, Lect. Notes in Comp. Sci. 52, pp. 282-300, 1977.

[5] Janssen, T. M. V. and Boas, P. van Emde : The expressive power of intensional logic in the semantics of programming languages, Lect. Notes in Comp. Sci. 53, pp. 303-311, 1977.

[6] Sawamura, H. : A logic of sequential programs based on Montague's intensional logic, Information Processing, Vol. 22, No. 3, IPSJ, pp. 216-224, 1981. (in Japanese).

[7] Sawamura, H.: Intensional logic as a basis of algorithmic logic, IIAS R. R. No. 13, 1981.

[8] Clifford, J. and Warren, D. S. : Formal semantics for time in databases, ACM Trans. on Database Systems, Vol. 8, No.2, pp. 214-254, 1983.

[9] Hobbs, J. R. : Making computational sense of Montague's intensional logic, Artificial Intelligence, Vol. 9, pp. 287-306, 1978.

[10] Weyhrauch, R. : FOL: A proof checker for first-order logic, AIM-235, Stanford University, 1977.

[11] Gordon, M. J., Milner, A. J. and Wadsworth, C. P. : Edinburgh LCF - A mechanized logic of computation, Lect. Notes in Comp. Sci. 78, 1979.

[12] Constable, R. L., Johnson, S. D. and Eichenlaub, C. D. : An introduction to the PL/CV2 programming logic, Lect. Notes in Comp. Sci. 135, 1981.

[13] Ketonen, J. and Weening, J. S. : EKL - An interactive proof checker, User's manual, Dept. of Comp. Sci., Stanford University, 1983.

[14] Sawamura, H. : Axiomatization of computer-oriented modal logic and decision procedure, Bulletin of Informatics and Cybernetics, Vol. 21, No. 3-4, pp. 57-66, 1985.

[15] Bowen, D. L. : Dec system-10 PROLOG USER'S MANUAL, version 3.43, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1983.

[16] Henkin, L. : A theory of propositional types, Fundamenta Mathematicae, Vol. 51, pp.323-344, 1963.

[17] Andrews, P. : A reduction of the axioms for the theory of propositional types, ibid., pp.345-350, 1963.

[18] Sawamura, H. : A proof checker for intensional logic, Research Association for Logical Grammar, 1985. (in Japanese).

[19] Sawamura, H : A proof constructor for intensional logic, IIAS R. R., 1985. (in preparation).

[20] Milner, R. : A theory of type polymorphism in programming, JACM, Vol. 17, pp. 348-375, 1978.

[21] Sawamura, H., T. Takeshima and A. Kato : Towards a descriptive language based on many-sorted equational logic, IIAS R. R., No, 42, 1983.

[22] Sawamura, H. and T. Takeshima : Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization, Proc. of the Symposium on Logic Programming, Boston, Ma., IEEE Computer Society, 1985.

[23] Hughes, G. E. and Cresswell, M. J. : An introduction to modal logic, Methuen and Co. Ltd., 1968.

[24] Ohnishi, M. and Matsumoto, K. : Gentzen method in modal calculi, I, II, Osaka Journal of Mathematics, Vol. 9, pp. 113-130, 1957, and Vol. 11, pp. 115-120, 1959.

[25] Slaght, R. L. : Modal tree constructions, Notre Dame J. of Formal Logic, Vol. 18, No. 4, pp. 517-526, 1977.

[26] Fitting, M. : Tableau methods of proof for modal logics, ibid., Vol. 8, No. 2, pp. 237-247, 1972.

[27] Wrightson, G. : A proof procedure for higher-order modal logic, Proc. 4th Workshop on Automated Deduction, pp. 148-154, 1979.

[28] Hindley, J. R., Lercher, B. and Seldin, J. P.: Introduction to combinatory logic, Cambridge Univ. Press, 1972.