数理解析研究所講究録
第396巻 1980年 199-235

Intensional Logic as a basis of Algorithmic Logic

Hajime Sawamura

International Institute for Advanced
Study of Social Information Science
(IIAS-SIS), Fujitsu Ltd.

Abstract

Intensional logic, which R. Montague developed in order to accomodate a wide variety of intensional locutions in natural language in a formal system, seems to be promising as an underlying logic for a logic of programs as well, since it can provide enough expressive power to describe modal concepts of programs and expressions with types.

In this article, we discuss problems of intensionality arising in programming languages and demonstrate that the intensional logic is able to give a useful logical basis for the deductive semantics of programming language and program verification by formalizing intensional Dijkstra logic (IDL), intensional Hoare logic (IHL) and intensional Manna and Pnueli logic (IMPL) within the general framework of intensional logic.

1. Introduction

There has been a considerable increase in the applications of modal logics recently, particularly to computational linguistics (1-3), program verification (4-11) and artificial intelligence (12-14).

This paper concerns an application of the intensional logic, a kind of higher-order modal logic, which R. Montague (15-16) developed in order to incorporate intensional locutions in natural language which has not had corresponding expressions in predicate calculus and modal logics, to algorithmic logic (or logic of programs). By algorithmic logic (or logic of programs) is understood a kind of logic in which a program in a specified language is treated as a mathematical object and the operation of language constructs and the properties of programs are axiomatically stipulated. So far, logics of programs have been presented with respective profitable features. In particular, Burstall (4), Schwarz (5), Ashcroft (6), Pratt (7), Kröger (8), Manna and Waldinger (9), Pnueli (10) and Manna and Pnueli (11) have observed close correspondences of syntactic and semantic concepts contained in modal logics and tense logic with those of programs, and demonstrated usefulness of these logics in the formalization of a logic of programs.

On the other hand, it was indicated by Janssen and Boas (17-18) that the problems of intensionality analogous to the

case of natural language could also be recognized in programming language as a language with relation to the descriptions of the semantics of assignment statements including variables with various types. As Russell's paradox in set theory was resolved in the theory of types, the problems of intensionality in the semantics of assignment statements was solved by distinguishing the 'intension' and 'extension' of an expression in intensional logic incorporated with a theory of types.

Taking into consideration with these results and the fact that Montague's intensional logic includes the concepts of modality and tense, we understand that it might be effective as an underlying logic of a logic of programs for both types of sequential and parallel programs, and furthermore it may be suggested that Mongague's logic can provide a semantical system as a common base for both programming and natural languages.

In this paper, we discuss problems of intensionality arising in programming languages and demonstrate that the intensional logic is able to give a useful logical basis for the deductive semantics of programming language and program verification by formalizing intensional Dijkstra logic (IDL), intensional Hoare logic (IHL) and intensional Manna and Pnueli logic (IMPL) within the general framework of intensional logic, following the principle of Janssen and Boas's intensional semantics of assignment statements.

2. The Intensionality in Languages

The reference problems of linguistic expressions have been a controversial subject in the logical analysis of languages for a long time. For example, consider the following inference

The temperature is ninety (1)

The temperature rises (2)

---

ninety rises (3)

Although this form of inference is permitted as valid in ordinary classical logics with equality, it is obvious that we can not admit this as a permissible inference in our real world. The problem similar to this arises in the semantics of programming language. The meaning of assignment statement x := t, where x is a simple variable and t is some arithmetic expression, is defined to be

$$[t/x]P \{ x := t \} P \qquad \text{(Hoare (19))},$$

$$P \{ x := t \} \exists z([z/x]P \wedge x = [z/x]t) \qquad \text{(Floyd (20))}.$$

as assignment axioms where P is a first order predicate and $[z/x]P$ denotes the expressions obtained from P by replacing all free occurrences of x in P by z. However, this rule of meaning yields undesirable results if applied to situations where x is not simple. For example, consider the following program

P := x ; x := x + 1 (4)

where x is a simple variable and p a pointer variable. If we take p = x as a postcondition, by applying the composition rule of Hoare we obtain

$$x = x + 1$$

which is a contradiction. As a next example, consider the following Hoare's assignment axioms which include array variables

$$1 = a(j)\ \{a(i) := 1\}\ a(i) = a(j) \qquad (5)$$

The precondition 1 = a(j) is not restrictive enough to the precondition about array elements (it should be i = j V 1 = a(j)).

As seen from these examples of natural language and programming language, the invalid inference, the contradiction and the less restrictive condition included in them are resulted from the simple replacements of subexpressions which constitute the sentence in natural language and the predicate in programming language by another expressions. In other words, the sentence (1) asserts only identity of denotations (called extension) of 'The temperature' and 'ninety', but the sentence (2) which include the same subexpression as (1) does not asserts something about the denotation of it, but pertains to the other denotation (called intension). From assertion about intension, nothing really follows as far as extensions are concerned. Thus the conclusion (3) is illegitimate. The same thing can be applied to (4) and (5). In (4), the post-condition p = x asserts that the denotation of a pointer

variable p is x itself, but an assignment x := x + 1 has the effect that makes the denotation of a variable x equal to the denotation of x plus 1. Therefore, it is not permitted that the latter x (extension) is substituted for the former x (intension). In (5), the sort of denotation of an array variable becomes a problem rather than the discrimination of extension and intension of an array variable. However, we consider it as a kind of reference problem in this paper.

Thus, when we give an account of the meaning of a linguistic expression, it is not enough to relate it to an object or set of objects, we must also provide a sense or concept or intension for the expression. This has been done in the direction of natural language fragment through intensional logic by Montague, and will be done in the direction of programming language through intensional logic with additional constructs.

3. Underlying Intensional Logic (IL)

In this chapter, we describe the tenseless version of Montague's intensional logic with additional constructs along the line of Gallin (21).

3.1 Syntax

In a general form, we introduce the intensional language with types so that intensionality of expressions in a programming language can be treated, although some parts of it are not used for our purpose.

Types. Let e(entity), t(truthvalue), s(state or possible world or point of reference) be three distinct objects. The set of types of IL is the smallest set of Type satisfying:

(1) $e, t \in$ Type,

(2) $\alpha, \beta \in$ Type imply $(\alpha,\beta) \in$ Type,

(3) $\alpha \in$ Type implies $(s,\alpha) \in$ Type.

We frequently write $\alpha\beta$ for (α,β) and $s\alpha$ for (s,α).

Primitive symbols. Primitive symbols consist of the following:

(1) variables: for each type α, a denumerable list of $x_\alpha^1, x_\alpha^2, x_\alpha^3, \ldots,$

(2) non-logical constants: for each type α, a denumerable list of $c_\alpha^1, c_\alpha^2, c_\alpha^3, \ldots,$

(3) special symbols: $\equiv, \lambda, \hat{}, \check{}, (,), [,], \{, \}, /, \rightarrow.$

We omit superscripts or subscripts as far as confusions

do not arise. By $\dot{CON}_\alpha(VAR_\alpha)$ is understood the set of constants (variables) of type α and $CON = \bigcup_{\alpha \in Type} CON_\alpha$, $VAR = \bigcup_{\alpha \in Type} VAR_\alpha$.

Terms. The set Tm_α of terms of IL of type α is inductively defined as follows:

(1) $CON_\alpha \subset Tm_\alpha$,

(2) $VAR_\alpha \subset Tm_\alpha$,

(3) $A \in Tm_{\alpha/\beta}$, $B \in Tm_\alpha$ imply $A(B) \in Tm_\beta$,

(4) $A \in Tm_\beta$, $x \in VAR_\alpha$ imply $\lambda x(A) \in Tm_{\alpha/\beta}$,

(5) $A, B \in Tm_\alpha$ imply $A \equiv B \in Tm_t$,

(6) $A \in Tm_\alpha$ implies $\hat{}A \in Tm_{s\alpha}$,

(7) $A \in Tm_{s\alpha}$ implies $\check{}A \in Tm_\alpha$,

(8) $A, B \in Tm_\alpha$, $p \in Tm_t$ imply $(p \rightarrow A, B) \in Tm_\alpha$,

(9) $A \in Tm_\alpha$, $x \in CON_{s\beta}$, $B \in Tm_\beta$ imply $\{B/\check{}x\}\, A \in Tm_\alpha$.

We introduce the sentential connectives, quantifiers and modal operators in IL by definition:

$T = (\lambda x_t x_t \equiv \lambda x_t x_t)$,

$F = (\lambda x_t x_t \equiv \lambda x_t T)$,

$\neg = \lambda x_t (F \equiv x_t)$,

$\wedge = \lambda x_t \lambda y_t (\lambda f_{tt}\ (f_{tt}\ x \equiv y) \equiv \lambda f_{tt}\ (fT))$,

$\supset = \lambda x_t \lambda y_t (x \wedge y \equiv x)$,

$\vee = \lambda x_t \lambda y_t (\neg x \supset y)$,

$\forall x_\alpha A = (\lambda x_\alpha A \equiv \lambda x_\alpha T)$,

$\exists x_\alpha A = \neg \forall x_\alpha \neg A$,

$LA_t = (\hat{}A_t \equiv \hat{}T)$,

$MA_t = \neg L \neg A_t$.

These definitions follow Henkin (22) and write $A \wedge B$

instead of $\wedge(A)(B)$, similarly for the other binary connectives.

3.2 Semantics

The terms of IL are interpreted in an intensional modal.

Frame. Let D and S be non-empty sets. By a frame based on D and S we understand the indexed family $(D_\alpha)_{\alpha \in Type}$ of sets, where

(1) $D_e = D$,

(2) $D_t = \{0, 1\}$,

(3) $D_{s\alpha} = D_\alpha{}^S = \{ f \mid f : S \to D_\alpha \}$,

(4) $D_{\alpha\beta} = D_\beta{}^{D_\alpha} = \{ f \mid f : D_\alpha \to D_\beta \}$.

Model. A model of IL based on D and S is a system $M = (D_\alpha, m)_{\alpha \in Type}$, where

(1) $(D_\alpha)_{\alpha \in Type}$ is a frame based on D and S,

(2) m(meaning function) is a mapping which assigns to each constant c_α a function from S into D_α; in symbols, $m(c_\alpha) \in D_\alpha{}^S$.

Assignment. An assignment a over M is a mapping on the set of variables of IL such that $a(x_\alpha) \in D_\alpha$ for every variables x_α of type α. Let a be an assignment, x a variable of type α and $d \in D_\alpha$. a(x/d) is an assigmment a' whose value a'(y) for a variable y is equal to d if y is x and a(y) otherwise.

Execution state. We have introduced the state abstractly as above. Here, in order to define meanings of programs in IL, we identify a state with an execution state which consists

of the values of all program variables (which are translated into constants in IL, see chapter 4) at a certain stage in the execution, that is, an(execution) state is a member of

$$\prod_{c \in CON_{se}} D_e \quad X \quad \prod_{c \in CON_{s(ee)}} D_{ee} \quad X \quad \prod_{c \in CON_{s(se)}} D_{se}$$

if a program operates over a domain D. Two states are equal iff all constants have equal values.

Value function. The value ${}_{s,a}V^{M}(A_\alpha)$ in M of a term A_α with respect to the state s and the assignment a is given recursively as follows (we suppress the superscript 'M'):

(1) ${}_{s,a}V(c_\alpha) = m(c_\alpha)(s)$, $c_\alpha \in CON$,

(2) ${}_{s,a}V(x_\alpha) = a(x_\alpha)$, $x_\alpha \in VAR$,

(3) ${}_{s,a}V(A_{\alpha\beta}(B_\alpha)) = {}_{s,a}V(A_{\alpha\beta})({}_{s,a}V(B_\alpha))$,

(4) ${}_{s,a}V(\lambda x_\alpha(A_\beta))$ = the function f on D_α whose value at $d \in D_\alpha$ is equal to ${}_{s,a'}V(A_\beta)$, where $a' = a(x_\alpha/d)$,

(5) ${}_{s,a}V(A \equiv B) = 1$ if ${}_{s,a}V(A_\alpha) = {}_{s,a}V(B_\alpha)$, and 0 otherwise,

(6) ${}_{s,a}V(\hat{}A_\alpha)$ = the function f on S whose value at $t \in S$ is equal to ${}_{t,a}V(A_\alpha)$,

(7) ${}_{s,a}V(\check{}A_{s\alpha}) = {}_{s,a}V(A_{s\alpha})(s)$,

(8) ${}_{s,a}V((p_t \rightarrow A_\alpha, B_\alpha)) = {}_{s,a}V(A_\alpha)$ if ${}_{s,a}V(p_t) = 1$, and ${}_{s,a}V(B_\alpha)$ otherwise,

(9) ${}_{s,a}V(\{ B_\beta / \check{}c_{s\beta} \} A_\alpha) = {}_{t,a}V(A_\alpha)$, where $t = \langle c \leftarrow {}_{s,a}V(B_\beta) \rangle s$, which denotes the state in which all constants except c have the same value as in the state s and the value of $\check{}c$ equals the value of the expression B_β in the state s.

Formula, Satisfied, True, Valid. A formula is a term of type t. A formula A is satisfied in M by a state s and an assignment a, symbolically, M, s, a $\models$ A iff $V^M_{s,a}(A) = 1$. A formula A is true in M, symbolically, M $\models$ A iff for every s and a, M, s, a $\models$ A. A formula A is valid, symbolically, $\models$ A iff for every M, M $\models$ A.

It is readily verified that the connectives, quantifiers and modal operators defined above have their usual meanings in any model M, that is, for example, M, s, a $\models$ (A) V (B) just in case either M, s, a $\models$ A or M, s, a $\models$ B, M, s, a $\models \forall x_\alpha A$ just in case M, s, a(x/d) $\models$ A for every $d \in D_\alpha$, and M, s, a $\models$ LA iff M, t, a $\models$ A for every $t \in S$, that is, the necessity operator L is an S5 operator of modal logic.

Free, Bound. An occurrence of a variable x_β in a term A_α is bound if it occurs within a part $\lambda x_\beta B_\gamma$, otherwise free.

3.3 Deductive theory

Axioms.

A_1. $g_{tt}(T) \wedge g_{tt}(F) \equiv \forall x_t(g_{tt}(x_t))$,

A_2. $x_\alpha \equiv y_\alpha \supset f_{\alpha t}(x_\alpha) \equiv f_{\alpha t}(y_\alpha)$,

A_3. $\forall x_\alpha(f_{\alpha\beta}(x_\alpha) \equiv g_{\alpha\beta}(x_\alpha)) \equiv (f_{\alpha\beta} \equiv g_{\alpha\beta})$,

A_4. $\lambda x_\alpha(A_\beta(x_\alpha))(B_\alpha) \equiv A_\beta(B_\alpha)$, where $A_\beta(B_\alpha)$ comes from $A_\beta(x_\alpha)$ by replacing all free occurrences of x_α by the term B_α, satisfying the conditions:

(1) no free occurrence of x_α in $A_\beta(x_\alpha)$ lies within a part $\lambda y(C)$ where y is free in B_α, and

(2) no free occurrence of x_α in $A_\beta(x_\alpha)$ lies within

the scope of ^, L, M, $\{E_\gamma/{}^\vee c_{s\gamma}\}$, or else

(2') ${}_{s,a}V(B_\alpha) = {}_{t,a}V(B_\alpha)$ for any state s, t $\in$ S(in other words, B is modally closed (see Gallin (21)).

A_5. $L({}^\vee f_{s\alpha} \equiv {}^\vee g_{s\alpha}) \equiv (f_{s\alpha} \equiv g_{s\alpha})$,

A_6. ${}^\vee\hat{}A_\alpha = A_\alpha$,

A_7-1. $\{E_\alpha/{}^\vee c_{s\alpha}\}c_\beta' \equiv c_\beta'$ for any constant c_β', including the case that c' is the same constant symbol as c and $\beta = (s,\alpha)$,

A_7-2. $\{E_\alpha/{}^\vee c_{s\alpha}\}{}^\vee c_{s\alpha} \equiv E_\alpha$; $\{E_\alpha/{}^\vee c'_{s\alpha}\}{}^\vee c_{s\gamma} \equiv c'_{s\gamma}$ for any constant $c'_{s\gamma}$ that is not the same constant symbol as $c_{s\gamma}$ and $(s,\gamma) \neq (s,\alpha)$,

A_7-3. $\{E_\alpha/{}^\vee c_{s\alpha}\}\, x_\beta = x_\beta$,

A_7-4. $\{E_\alpha/{}^\vee c_{s\alpha}\}\, A_{\beta\gamma}\ (B_\beta) \equiv \{E_\alpha/{}^\vee c_{s\alpha}\}\, A_{\beta\gamma}\ (\{E_\alpha/{}^\vee c_{s\alpha}\}\, B_\beta)$,

A_7-5. $\{E_\alpha/{}^\vee c_{s\alpha}\}\lambda x_\beta(A_\gamma) \equiv \lambda x_\beta(\{E_\alpha/{}^\vee c_{s\alpha}\}\, A_\gamma)$, provided that x_β does not occur in E_α (otherwise we take an alphabetical variant of $\lambda x_\beta(A_\gamma)$,

A_7-6. $\{E_\alpha/{}^\vee c_{s\alpha}\}(A_\beta \equiv B_\beta) \equiv (\{E_\alpha/{}^\vee c_{s\alpha}\}\, A_\beta \equiv \{E_\alpha/{}^\vee c_{s\alpha}\}\, B_\beta)$,

A_7-7. $\{E_\alpha/{}^\vee c_{s\alpha}\}\, \hat{}A_\beta \equiv \hat{}A_\beta$,

A_7-8. $\{E_\alpha/{}^\vee c_{s\alpha}\}\{E'_\alpha/{}^\vee c_{s\alpha}\}\, A_\beta \equiv \{\{E_\alpha/{}^\vee c_{s\alpha}\}\, E'_\alpha/{}^\vee c_{s\alpha}\}\, A_\beta$,

A_7-9. $\{E_\alpha/{}^\vee c_{s\alpha}\}(p_t \rightarrow A_\beta, B_\beta) \equiv (\{E_\alpha/{}^\vee c_{s\alpha}\}\, p_t \rightarrow \{E_\alpha/{}^\vee c_{s\alpha}\}A_\beta, \{E_\alpha/{}^\vee c_{s\alpha}\}\, B_\beta)$,

Rule of Inference. From $A_\alpha \equiv A'_\alpha$ and the formula B_t to infer the formula B'_t, where B'_t comes from B_t by replacing one occurrence of A_α by the term A'_α.

Proof, Provavle, Theorem. A proof of IL is a sequence

of formulas each of which is either an axiom or else is obtainable from earlier formula by the inference rule. A formula A_t is provable in IL, or a theorem of IL, and we write $\vdash A_t$ if it is the last line of a proof.

Theorem 1. (Soundness theorem) $\vdash A$ implies $\models A$.

The proof is omitted.

IL includes the ordinary laws of sentential and predicate logic, together with the S5 modal laws. We list some of them,

T_1. $\vdash A$, where A comes from a tautology by uniform substitution of formulas of IL for free variables,

T_2. $\vdash \forall x\, A(x) \supset A(B_\alpha)$, where $A(x)$ and B_α satisfy the conditions of Axiom A_4,

T_3. $\vdash A \supset B$ implies $\vdash A \supset \forall x_\alpha B$, where x_α is any variable not free in A (Generalization),

T_4. $\vdash A \supset B$ and $\vdash A$ imply $\vdash B$ (Modus ponens),

T_5. $\vdash LA \supset A$,

T_6. $\vdash L(A \supset B) \supset (LA \supset LB)$,

T_7. $\vdash MA \supset LMA$ (S5 axiom),

T_8. $\vdash L\forall x_\alpha A \equiv \forall x_\alpha LA$ (Barcan formula),

T_9. $\vdash A$ implies $\vdash LA$ (Necessitation).

3.4 Expressiveness of IL

The intensional language described in this chapter have two constructs augmented to Montague's intensional logic. The term $(P_t \rightarrow A_\alpha, B_\alpha)$ is used to represent the denotation of array variable as a function. This term has the form of McCarthy's conditional expression (25). The operator $\{E_\alpha/{}^\vee c_{s\alpha}\}$, which is called state switcher by Janssen and Boas and have almost the same properties as the ordinary substitution operator, is used to represent the semantics of assignment statement as an effect of it to a state. The properties of state switcher are introduced as axioms A_7-1 $\sim$ A_7-9 in the same way as other logical operators. $\hat{}$(called cap or up operator) and $^\vee$(called cup or down operator) are used to represent the intension, extension of expressions respectively, for example, the extension of a simple variable in a programming language is the value of it in a particular state of possible execution states and its intension is a function which for any possible state yields the corresponding value. The extension of an assertion for proving properties of programs is the truthvalue of it in a particular state of possible execution states and its intension is a function which for any possible state yields the corresponding truthvalue, that is, a Boolean function on the set of possible states. Predicates and quantifiers in IL are used to represent properties of a single state and modalities, L, M to represent properties of the execution leading from one state to another.

4. Intensional Dijkstra Logic (IDL)

The intensional logic in chapter 2 has a sufficient expressive power for expressing the various kinds of typed variables appearing in a programming language and their meanings. In this chapter, we consider a smallest deterministic programming language of which intensionality comes into question and its semantics in terms of Dijkstra's notion (23) of predicate transformers, a formalism for specifying the semantics of a program construct by the transformation that the construct defines. And the predicate transformers are well described in the intensional logic. The following descriptions are given informally to such an extent that ambiguities do not arise.

4.1 A simple programming language (SPL)

Symbols :

(1) constants: numerical constants, true, false, etc.,

(2) variables : simple variables x, y, ..., array variables a, b, ..., pointer variables p, q, ...,

(3) logical symbols : $\neg$, $\wedge$, $\vee$,

(4) arithmetical and relational symbols : +, -, x, $\div$, $>$, $\geqq$, $<$, $\leqq$, =,

(5) program symbols : ;, :=, if, then, else, fi, while, do, od, go to, null,

(6) label symbols,

(7) auxiliary symbols (,), [,].

Arithmetic expressions E and Boolean expressions F :

These are not further prescribed here, but it is assumed that Boolean expressions are always truth-valued and arithmetic expressions have no side effects.

Programs Prog :

(1) null statement : null $\in$ Prog,

(2) assignment statement : if e, f $\in$ E, x is a simple variable, a is a array variable and p, q are a pointer variables, then

(2.1) x := f $\in$ Prog,

(2.2) a(e) := f $\in$ Prog,

(2.3) p := x $\in$ Prog,

(2.4) p := a(e) $\in$ Prog,

(2.5) p := q $\in$ Prog,

(3) composition : if S_1, S_2 $\in$ Prog, then S_1; S_2 $\in$ Prog,

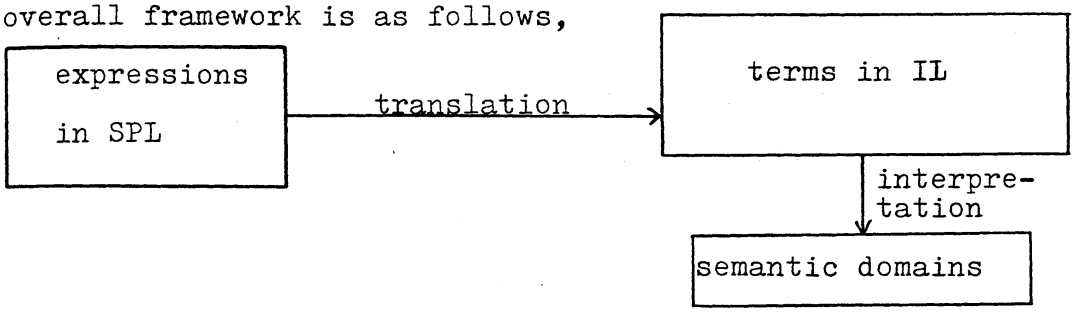
(4) conditional : if S, S_1, S_2 $\in$ Prog and p $\in$ F, then if p then S_1 else S_2 fi, if p then S fi $\in$ Prog,

(5) iteration : if S $\in$ Prog and p $\in$ F, then while p do S od $\in$ Prog.

4.2 Semantics

The semantics of SPL based on the IL is given in such a way that R. Montague (15-16) gave the semantics of a fragment of natural language based on intensional logic. That is, each expression exp in a program written in SPL is translated into the term exp' in IL by applications of translation rules to

the syntactic structures in SPL. The exp' is called a translation of exp and ' is called a translation function. Of cource, a meaning of exp' is determined by the model of IL. Thus, the overall framework is as follows,



4.3 Translation Rules

Translation of symbols in SPL :

Logical symbols, arithmetic and relational symbols, auxiliary symbols and equality symbols in SPL translate into the corresponding symbols in IL. Numerical constants, simple variables, array variables and pointer variables translate into the non-logical constants with types e, (s,e), (s, (e, e)), (s, (s e)) respectively. As the translated symbols, we use the same symbols as those in SPL as far as confusions do not arise.

Translation of arithmetic and Boolean expressions in SPL :

The translations of an arithmetic expression and a Boolean expression are obtained from their subexpressions. Simple variables x and array elements a(e) occurring in the righthand sides of the assignment statements (2.1) and (2.2) and in Boolean expressions, and a pointer variable q in the righthand side of (2.5) translate into ˇx, ˇa(e'), ˇq respectively, a simple variable x and an array element a(e) occurring in the righthand sides of the assignment statements (2.3) and (2.4)

translate into x, a(e') respectively, where e' denotes the translated symbol or expression which is obtained through the same way of translation as those occurring in the righthand sides of the assignment statements (2.1) and (2.2).

Translation of programs :

We call a term of type (s,t) a state predicate, and use it as a specification for proving that a program is correct. The reason that the type of the specification of a program is (s,t), not simply t as in usual logics of programs lies in the time or state dependency of truthhood of a proposition (a specification) according to the execution states of a program. Therefore, in the intensional framework of programs we employ a state predicate for the specification of a program to explicitly express that fact.

In intensional Dijkstra logic (IDL), the semantics of programs is dealt with by translating them into predicate transformers (PT's) (in Dijkstra's notation wp), which are terms in IL of type ((s,t), (s,t)) and have the form $\lambda P(A)$, where $P \in VAR_{st}$ and A is a term of type (s,t) constructed from each program construct. That is, let Q be an arbitrary state predicate expressing an postcondition. $\lambda P(A)$ (Q) is the state predicate expressing the weakest precondition such that the program construct terminates and produces a final state satisfying $^{\vee}Q$. As well as Montague's method, the translation rules in the following are in one to one correspondence with the syntactic rules in the definition of Prog. Therefore,

the translation of a compound expressions is determined by the translation of its subexpressions.

(1) null statement : For 'null', its semantics is defined to be

$$\lambda P(P)$$

(2) assignment : For the case of 'x := f' where x denotes a simple variable or pointer variable, its semantics is defined to be

$$\lambda P\hat{}(\{f'/\check{}x\}\check{}P),$$

for the case of 'a(e) := f',

$$\lambda P\hat{}(\{\lambda n(n=e' \rightarrow f',\ \check{}a(n))/\check{}a\}\check{}P)$$

(and in case of an array with higher dimension, this predicate transformer is used repeatedly).

(3) composition : For 'S_1; S_2' with translations PT_1, PT_2 respectively, its semantics is defined to be

$$\lambda P\hat{}(PT_1(PT_2(P))).$$

(4) conditional : For 'if p then S_1 else S_2 fi' with translations p', PT_1, PT_2 respectively, its semantics is defined to be

$$\lambda P\hat{}((p'\wedge\check{}PT_1(P))\vee(\neg p'\wedge\check{}PT_2(P)),$$

for 'if p then S fi' with translations p' PT respectively, its semantics is defined to be

$$\lambda P\hat{}(p'\wedge\check{}PT(P)\ \vee\neg p'\wedge\check{}P).$$

(5) iteration : For 'while p do S od' with translations p', PT respectively,

let

$PT^0 = \lambda P\hat{}(\check{}P \wedge \neg p')$,

$PT^{j+1} = \lambda p\hat{}(p' \wedge \check{}PT(PT^j(P)))$, then its semantics is defined to be

$$\lambda P\hat{}(\exists\, j(\check{}PT^j(P))).$$

4.4 Program verification

We have defined the formal semantics for program constructs which is necessary to prove that a program is correct. Under these definitions, we introduce the concepts of termination, consistency, correctness, a method for their verification which are intensional versions of those introduced by R. Yeh (24), as follows: Let PT_S be a term constructed from a program S and P, Q be state predicates.

Termination : A program S terminates with respect to P iff $\vdash \check{}P \supset \check{}PT_S(\hat{}true)$,

Consistency (Partial correctness) : A program is consistent with (P,Q) iff $\vdash \check{}P \supset (\check{}PT_S(\hat{}true) \supset \check{}PT_S(Q))$,

Correctness (total correctness) : A program is correct with respect to (P,Q) iff $\vdash \check{}P \supset \check{}PT_S(Q)$.

4.5 Examples of intensional predicate transformers

(1) Let p := x ; x := x + 1 be a program and the corresponding predicate transformers of the two assignment statements be

$PT_1 = \lambda P\hat{}(\{ x/\check{}p \}\ \check{}P)$,

$PT_2 = \lambda P\hat{}(\{\check{}x + 1/\check{}x\}\ \check{}P)$, respectively.

Consider as a state predicate after the execution $\hat{}(\check{}p \equiv x)$.

Then, $\lambda P(PT_1(PT_2(P)))(\hat{}(\check{}p \equiv x))$

$= \lambda P\hat{}(\{ x/\check{}p \}\ \check{}P)\check{}\hat{}(\check{}p \equiv x)$

$= \hat{}(\{ x/\check{}p \}\ (\check{}p \equiv x))$

$= \hat{}(x \equiv x)$.

This is a correct solution to the problem of pointer in chapter 2.

(2) Let 'while $x \neq 0$ do $x := x - 1$ od' be a program and $\hat{}(\check{}x \equiv 0)$ a state predicate.

$(x := x - 1)' = \lambda P\hat{}(\{\check{}x - 1/\check{}x\}\ \check{}P)$.

$(x \neq 0)' = (\check{}x \not\equiv 0)$.

Hence, $PT^0(\hat{}(\check{}x \equiv 0)) = \hat{}(\check{}x \equiv 0 \wedge \check{}x \equiv 0) = \hat{}(\check{}x \equiv 0)$.

$PT^1(\hat{}(\check{}x \equiv 0)) = \hat{}((\check{}x \not\equiv 0) \wedge \check{}\lambda P\hat{}(\{\check{}x - 1/\check{}x\}\ \check{}P)$
$(\hat{}(\check{}x \equiv 0))) = \hat{}(\check{}x \not\equiv 0 \wedge \check{}x \equiv 1) = \hat{}(\check{}x \equiv 1)$.

$PT^2(\hat{}(\check{}x \equiv 0)) = \hat{}(\check{}x \equiv 2)$.

Generally, $PT^j(\hat{}(\check{}x \equiv 0)) = \hat{}(\check{}x \equiv j)$.

Hence, we obtain $\hat{}(\exists j(\check{}x \equiv j))$ as a state predicate before the execution. Since $\vdash \check{}\hat{}(\check{}x > 0) \supset \check{}\hat{}(\exists j(\check{}x \equiv j))$, the iteration program is correct with respect to the intensional specification $(\hat{}(\check{}x > 0), \hat{}(\check{}x \equiv 0))$.

5. Intensional Hoare Logic (IHL)

In this chapter, we consider the intensional version of Hoare logic (19) which defines language constructs in terms of how programs containing them could can be proved correct, instead of in terms of how they were to be executed. It consists of a logical system of axiom schemata and inference rules. The primitive formulas in this system are of the form $P\{S\}Q$, where P and Q are state predicates like in Dijkstra logic, and are interpreted as below.

Definition 1 : $M_A = \{(s,t) \mid$ program A, Started in a state s, terminates in a state t$\}$.

Definition 2 : We extend the definition of intensional language as follows, If $P, Q \in Tm_{st}$ and A is a program, then $P\{A\}Q \in Tm_t$.

Definition 3 : We extend the interpretation ${}_{s,a}V^M$ for $P\{A\}Q$ as follows,

${}_{s,a}V^M(P\{A\}Q) = \begin{cases} T \text{ if } {}_{s,a}V^M(\check{}P) = T \text{ and } (s,t) \in M_A \text{ imply } {}_{t,a}V^M(\check{}Q) = \\ F \text{ otherwise.} \end{cases}$

Definition 4 : A program A is (Partially) correct with respect to a pair of an input state predicate P and an output state predicate Q (or input-output specifications (P,Q))iff $\vdash P\{A\}Q$.

5.1 Proof system

We use the symbols as follows,

P, Q, R stand for state predicates,

S, S_1, S_2 stand for programs,

p stands for a Boolean expression,

x stands for a simple variable or a pointer variable,

a stands for an array variable,

e, f stand for arithmetic expressions,

stands for a translation function.

Axioms

Assignment axioms :

$$\hat{}(\{f'/\check{}x\}\check{}P)\ \{\ x := f\ \}\ P$$

$$\hat{}(\{\lambda n(n=e' \rightarrow f', \check{}a(n))/\check{}a\}\check{}P)\ \{a(e) := f\}\ P$$

Theorems : P for all P such that $\vdash p$

Inference rules

Consequence : $$\frac{\check{}P \supset \check{}Q \quad Q\{S\}R}{P\{S\}R}\ ,\quad \frac{P\{S\}Q \quad \check{}Q \supset \check{}R}{P\{S\}R}$$

Composition : $$\frac{P\{S_1\}Q \quad Q\{S_2\}R}{P\{S_1;\ S_2\}R}$$

Null : $$\frac{\check{}P \supset \check{}R}{P\{\text{Null}\}R}$$

Conditional : $$\frac{\hat{}(\check{}P \wedge p')\{S_1\}Q \quad \hat{}(\check{}P \wedge \neg p')\{S_2\}Q}{P\{\text{if } p \text{ then } S_1 \text{ else } S_2 \text{ fi}\}Q}\ ,$$

$$\frac{\hat{}(\check{}P \wedge p')\{S\}Q \quad \check{}P \wedge \neg p' \supset \check{}Q}{P\{\text{if } p \text{ then } S \text{ fi}\}Q}$$

Iteration : $$\frac{\hat{}(\check{}P \wedge p')\{S\}P}{P\{\text{while } p \text{ do } S \text{ od}\}\hat{}(\check{}P \wedge \neg p')}$$

5.2 Properties of IHL

Theorem 2 (Soundness theorem). If $\vdash P\{A\}Q$, then $\models P\{A\}Q$.

Proof. First we define by cases the final state FS(A,s) of a program A relative to a model as follows :

(1) FS(NULL, s) = s

(2) If x := e is an any type of assignment statement, then
FS(x := e, s) $= \langle x' \leftarrow {}_{s,a}V(e') \rangle s$

(3) $FS(S_1; S_2, s) = FS(S_2, FS(S_1, s))$

(4) FS(if p then S_1 else S_2 fi, s)
$$= \begin{cases} FS(S_1,s) & \text{if } {}_{s,a}V(p') = T \\ FS(S_2,s) & \text{otherwise} \end{cases}$$

(5) FS(while p do S od, s)
$$= \begin{cases} FS(\text{while } p \text{ do } S \text{ od}, FS(S,s)) & \text{if } {}_{s,a}V(p') = T \\ s & \text{otherwise} \end{cases}$$

We will only show that the axioms are valid and that the iteration rule preserves truth since the other parts of the rules of inference are easy. We will fix any model M and first consider any type of assignment axioms. Suppose that for any state s, $s \models \{e'/\check{}x\}\check{}P$, then for any a, ${}_{s,a}V(\{e'/\check{}x\}\check{}P) = {}_{t,a}V(\check{}P)$, where $t = \langle x' \leftarrow {}_{s,a}V(e') \rangle s$. From $(s,t) \in M_{x := e}$, we obtain $t \models \check{}P$, thus, $\hat{}(\{e'/\check{}x\}\check{}P)\{x := e\}P$ is valid,

Next we will consider the iteration rule. For a state s

such that $s \not\models p'$, the upper formula of iteration rule is vacuously true and the lower formula of it is also true since $s \models {}^{\vee}P$ and $s \not\models p'$ imply $s \models {}^{\vee}P \wedge \neg p'$. Therefore, for any state s such that $s \models p'$, suppose that $s \models {}^{\wedge}({}^{\vee}P \wedge p')\{ S \} P$. Then,

$s \models {}^{\vee}P \wedge p'$ imply $t \models {}^{\vee}P$ for a terminating state t such that $(s,t) \in M_S$, (1)

Suppose that the lower formula does not hold. Then, for some initial state s_0 and the final state s_n such for $n > 0$, $(s_0, s_n) \in M_{\text{'while p do S od'}}$

$s_0 \models {}^{\vee}P$ (2) and

$s_n \not\models {}^{\vee}P \wedge \neg p'$ (3).

By the termination of the iteration statement, there exists a finite ensuing sequence of state-pairs whose first element denotes an initial state of S and whose second element denotes the corresponding final state of S,

$(s_0, s_1), (s_1, s_2), \ldots\ldots, (s_{n-1}, s_n)$, such that

$s_i \models p'$ for $0 \leq i \leq n-1$ (4) and

$s_n \not\models p'$ (5).

By repeated applications of (1) using (2) and (4), we obtain

$s_n \models {}^{\vee}P$ (6).

However, we have from (3) and (5), $s_n \not\models {}^{\vee}P$. This contradicts to (6). Q.E.D.

IHL is a functional variant of pure Hoare logic, that is, pure Hoare logic is obtained by eliminating the symbols $^{\wedge}$, $^{\vee}$, $'$ in the proof system of IHL. Thus,

Theorem 3. If $\vdash P\{A\}Q$ in pure Hoare logic, then $\vdash P\{A\}Q$ in IHL.

5.3 Proof Example

We show that the following program is correct with respect to $(\hat{}true, \hat{}(\check{}p \equiv i \wedge \check{}i \equiv 0 \vee \check{}p \equiv j \wedge \check{}j \equiv 4))$, which reduces to $(\hat{}true, \hat{}(\check{}\check{}p \equiv 0 \vee \check{}\check{}p \equiv 4))$.

Program : i := 0 ;

j := 4 ;

if random $<$ 0.5 then p := i else p := j fi

Correctness proof proceeds as follows :

1. $\hat{}true\{ i := 0 \}\hat{}(\check{}i \equiv 0)$ (axiom)
2. $\hat{}(\check{}i \equiv 0)\{ j := 4 \}\hat{}(\check{}i \equiv 0 \wedge \check{}j \equiv 4)$ (axiom)
3. $\check{}i \equiv 0 \wedge \check{}j \equiv 4 \wedge \check{}random \geq 0.5 \supset \check{}random \geq 0.5 \wedge \check{}j \equiv 4$ (theorem)
4. $\check{}i \equiv 0 \wedge \check{}j \equiv 4 \wedge \check{}random < 0.5 \supset \check{}random < 0.5 \wedge \check{}i \equiv 0$ (theorem)
5. $\hat{}(\check{}i \equiv 0 \wedge \check{}j \equiv 4 \wedge \check{}random \geq 0.5)\{ null \}\hat{}(\check{}random \geq 0.5 \wedge \check{}j \equiv 4)$ (from 3)
6. $\hat{}(\check{}i \equiv 0 \wedge \check{}j \equiv 4 \wedge \check{}random < 0.5)\{ null \}\hat{}(\check{}random < 0.5 \wedge \check{}i \equiv 0)$ (from 4)
7. $\hat{}(\check{}i \equiv 0 \wedge \check{}j \equiv 4 \wedge \check{}random \geq 0.5)\{p := j\}\hat{}(\check{}random < 0.5 \wedge \check{}p \equiv i \wedge \check{}i \equiv 0 \vee \check{}random \geq 0.5 \wedge \check{}p \equiv j \wedge \check{}j \equiv 4)$ (from 5)
8. $\hat{}(\check{}i \equiv 0 \wedge \check{}j \equiv 4 \wedge \check{}random < 0.5)\{p := i\}\hat{}(\check{}random < 0.5 \wedge \check{}p \equiv i \wedge \check{}i \equiv 0 \vee \check{}random \geq 0.5 \wedge \check{}p \equiv j \wedge \check{}j \equiv 4)$ (from 6)

9. $\hat{}\mathrm{true}\{$ i := 0 ; j := 4 ; if random $<$ 0.5 then p := i else p := j $\}$ $\hat{}(\check{}\mathrm{random} < 0.5 \wedge \check{}p \equiv i \wedge \check{}i \equiv 0 \vee \check{}\mathrm{random} \geq 0.5 \wedge \check{}p \equiv j \wedge \check{}j \equiv 4)$ (from 7, 8)

10. $\check{}\mathrm{random} < 0.5 \wedge \check{}p \equiv i \wedge \check{}i \equiv 0 \vee \check{}\mathrm{random} \geq 0.5 \wedge \check{}p \equiv j \wedge \check{}j \equiv 4 \supset \check{}p \equiv i \wedge \check{}i \equiv 0 \vee \check{}p \equiv j \wedge \check{}j \equiv 4$ (theorem)

11. $\hat{}\mathrm{true}\{$ i := 0 ; j := 4 ; if random $<$ 0.5 then p := i else p := j fi $\}$ $\hat{}(\check{}p \equiv i \wedge \check{}i \equiv 0 \vee \check{}p = j \wedge \check{}j \equiv 4)$ (from 9, 10).

6. Intensional Manna and Pnueli Logic (IMPL)

In this chapter, we consider Manna and Pnueli's modal logic (11) within the intensional logic, and give the proof system for the verification of a program including array and pointer variables in addition to simple variables, based on their 'sometime system' which is an appropriate formalization of the Intermittent-Assertion method (9). Also in this logic, the concept of 'state' corresponds to an execution state of a program, furthermore, the concept of 'accessibility relation' corresponds to the relation of derivability between states during execution.

State. In chapter 3, we defined the concept of state to be an execution state. However, here, we employ a little different definition. An (execution) state consists of the pair of the label name attached to a program statement and the values of all program variables at a certain stage in the execution, that is, an (execution) state has the general structure,

$$s = (l, \eta),$$

where $l \in \{l_0, l_1, \ldots, l_e\}$, a set of labels, labeling every statement of a program in which l_0, l_e denote the single entry label, the single exit label respectively, and

$$\eta \in \prod_{c \in CON_{se}} D_e \times \prod_{c \in CON_{s(ee)}} D_{ee} \times \prod_{c \in CON_{s(se)}} D_{se}$$ if a program operates over a domain D.

Distinguished propositional constants. Suppose that we have a set of special propositional constants, atl_0, atl_1, ..., atl_e, each corresponding to one of the labels introduced above, which in fact is a subset of constants with type t in IL, and suppose that

$_{s,a}V(atl_i) = T$ for such a state that $s = (l, \eta)$ (iff $m(atl_i)(s) = T$) iff $l_i = l$.

With these conventions we can express meaningful properties of programs and their execution in IL, for example, invariance properties having a form, $A \supset LB$ such as partial correctness, clean behavior, gloval invariant, mutual exclusion, deadlock freedom, eventuality properties having a form, $A \supset MB$ such as total correctness, general eventuality, accessibility, responsiveness.

6.1 Program and its translation

In IMPL, we consider the same programming language as in IDL, IHL, but in a directed graph representation.

Assignment statement.

If e, f are arithmetic expressions, then (1) $x \leftarrow f$, (2) $a(e) \leftarrow f$, (3) $p \leftarrow x$, (4) $p \leftarrow a(e)$, (5) $p \leftarrow q$ are assignment statements. Assume that the program is represented as a directed graph whose nodes are the program locations or labels, and whose edges represent transitions between the labels. A transition is an instruction of the general forms;

(1') $c \rightarrow (x \leftarrow f)$,

(2') $c \rightarrow (a(e) \leftarrow f)$,

(3') $c \rightarrow (p \leftarrow x)$,

(4') $c \rightarrow (p \leftarrow a(e))$,

(5') $c \rightarrow (p \leftarrow q)$,

where c is a condition (may be the trivial condition true) under which the transition should be taken and executed. We assume that all the conditions on transitions departing from any node are mutually exclusive and exhaustive (see Manna (26) for a precise definition of programs with simple variables).

Translation of symbols. Same as in IDL, IHL.

Translation of arithmetic and Boolean expressions. Same as in IDL, IHL.

Translation of Programs. Programs translate into a set of axioms reflecting the properties of the domain and the structure of the program under consideration, which is used in the proof system for proving properties of programs.

6.2 Proof system

Our proof system consists of three parts like in Manna and Pnueli's proof system.

General part. This part is the underlying IL introduced in chapter 3.

Proper part. This part specifies all the needed properties of the domain manipulated by programs as proper axioms or axiom schema. An essential axiom for every domain should be induction axiom. Thus the induction principle for natural numbers is.

$$\vdash \forall A_{et}(A(0) \wedge \forall x_e(A(x) \supset A(x+1)) \supset \forall x A(x))$$

Local part. The general and proper parts represent the general framework needed for our reasoning. In this part we introduce the deductive semantics of programs in the form of axioms translated into the formulas of IL. As a result, semantically the local axioms generated in such a way constrain the set of states and indirectly represent the program in the proof system by characterizing the possible transitions between states under program control.

Frame axiom. $\vdash A \supset LA$, where A is any formula which does not contain those constants that are translations of program variables (in other words, A is modally closed)(in fact, this is a theorem in IL).

Location Axiom. $\vdash \overset{e}{\underset{i=0}{V}} atl_i = T$, V denotes exclusive OR.

For any transition in a program represented as a directed graph;

$$l_i \xrightarrow{c \rightarrow t} l_j$$

where l_i, l_j denote labels, c denotes a Boolean expression, and t is one of the assignment statements, we generate the forward axiom schema (strongest postcondition (20)) F_i depending on the types of assignment statements (i), $1 \leq i \leq 5$ as follows:

F_1, F_3, F_4, F_5 : $\vdash atl_i \wedge c' \wedge {}^{\vee}A_{st} \supset M(atl_j \wedge \exists z(\{z/{}^{\vee}x\}{}^{\vee}A_{st} \wedge {}^{\vee}x \equiv \{z/{}^{\vee}x\}f'))$,

where x denotes the translated symbol of the lefthand side of assignment, f denotes the righthand side of it and A_{st} denotes an arbitrary state predicate.

F_2 : $\vdash atl_i \wedge c' \wedge {}^{\vee}A_{st} \supset M(atl_j \wedge \exists z(\{z/{}^{\vee}a\}\,{}^{\vee}A_{st} \wedge {}^{\vee}a \equiv \{z/{}^{\vee}a\}\,(\lambda n(n \equiv e' \rightarrow f', {}^{\vee}a(n)))$.

And similarly the backward axiom schema (weakest precondition (19)) B_i as follows :

B_1, B_3, B_4, B_5 : $\vdash atl_i \wedge c' \wedge \{f'/{}^{\vee}x\}\,{}^{\vee}A_{st} \supset M(atl_j \wedge {}^{\vee}A_{st})$, with the same conventions,

B_2 : $\vdash atl_i \wedge c' \wedge \{\lambda n(n \equiv e' \rightarrow f', {}^{\vee}a(n))/{}^{\vee}a\}\,{}^{\vee}A_{st} \supset M(atl_j \wedge {}^{\vee}A_{st})$.

We can easily see that these axiom schemata are functional variant of local axioms by Manna and Pnueli. Thus, Manna and Pnueli's modal logic of programs, 'Sometime system', has been embedded within the intensional logic with these additional axioms. Similarly, it would be possible to develop 'Nexttime system' of Manna and Pnueli (11) within IL with next operator.

Derived rules. The following derived rules hold in IL.

D_1. $\vdash A \supset B$ implies $\vdash LA \supset LB$ (LL-generalization),

D_2. $\vdash A \supset B$, $\vdash B \supset MC$ and $\vdash C \supset D$ imply $\vdash A \supset MD$ (Consequence),

D_3. $\vdash A \supset MB$ and $\vdash B \supset MC$ imply $\vdash A \supset MC$ (Concatenation),

D_4. $\vdash A \supset MB$ implies $\vdash A \wedge C \supset M(B \wedge C)$, where C satisfies the same condition as Frame axiom (Frame rule).

6.3 Example of local axiom

$l_i \xrightarrow{true \rightarrow (a(i) \leftarrow 1)} l_j$. Assume that after the assignment, a state predicate ${}^{\wedge}({}^{\vee}a({}^{\vee}i) \equiv {}^{\vee}a({}^{\vee}j))$ holds. Then we have as an instance of local backward axiom schema B_2,

$\vdash atl_i \wedge true \wedge \{\lambda n(n \equiv \check{}i \rightarrow 1, \check{}a(n))/\check{}a\}\ \check{}\hat{}(\check{}a(\check{}i) \equiv \check{}a(\check{}j)) \supset M(atl_j \wedge \check{}\hat{}(\check{}a(\check{}i) \equiv \check{}a(\check{}j)))$, which reduces to

$\vdash atl_i \wedge (\check{}i \equiv \check{}j \vee \check{}a(\check{}j) \equiv 1) \supset M(atl_j \wedge (\check{}a(\check{}i) \equiv \check{}a(\check{}j)))$.

This is a correct solution in IMPL, to the array problem in chapter 2.

7. Concluding Remarks

We have formalized intensional Dijkstra logic (IDL), intensional Hoare logic (IHL) and intensional Manna and Pnueli logic (IMPL) in order to demonstrate that Montague's intensional logic is able to give a useful logical basis for the deductive semantics of programming language and program verification.

The main contributions of intensional logic to logics of programs seem to be the provision of

1) modal concepts of programs which are concerned with time or state dependency of propositions,
2) expressions in the type hierarchy.

Although we confined ourselves to IDL, IHL and IMPL in this paper, the method for semantics based on a general and uniform framework of IL is applicable to other logics of programs (algorithmic logics) as well, as far as the semantics of assignments is given by reducing a semantic problem (the meaning of an assignment) to a syntactic solution (substitution in logic).

Effective proof procedure for intensional logic is worthwhile to be considered. Here, we will indicate two feasible ways for effective proof procedure for intensional logic. One is to implement it as an equational calculus as described by Robinson (27). The other is to implement it by representing the intensional language syncategorematically and using the

proof procedure for higher-order modal logic based on tableau method (28).

Acknowledgements.

I wish to thank Mr. T. Maeda and Dr. Y. Momouchi of Hokkaido Univ. for various helpful discussions and useful suggestions on the paper.

I would like to express my gratitude to Prof. T. Kitagawa, the director of IIAS-SIS for his invaluable advice and encouragement, and my colleagues for stimulating discussions.

References

1. C. Schwind : A formalism for the description of question answering systems, Lect. Notes in Comp. Sci. 63, Natural language communication with computers, Springer, 1978.

2. J. R. Hobbs and S. J. Rosenschein : Making computational sense of Montague's intensional logic, Artificial Intelligence, Vol. 9, No. 3, 1977.

3. T. M. W. Janssen : Logical investigations on PTQ arising from programming requirements, Mathematisch centrum, ZW 117/78, Amsterdam, 1978.

4. R. M. Burstall : Program proving as hand simulation with a little induction, IFIP 74, 1974.

5. J. Schwarz : Event based reasoning - A system for proving correct termination of programs, in Automata, Languages and Programming, edited by S. Michaelson and R. Milner, Edinburgh Univ. Press, 1976.

6. E. A. Ashcroft and W. W. Wadge : Lucid - A formal system for writing and proving programs, SIAM J. on Computing, Vol. 5, No. 3, 1976.

7. V. R. Pratt : Semantical considerations on Floyd-Hoare logic, 17th IEEE Symp. on Found. of Comp. Sci., 1976.

8. F. Kröger : LAR : A logic of algorithmic reasoning, Acta Informatica, Vol. 8, Fasc. 3, 1977.

9. Z. Manna and R. Waldinger : Is 'Sometime' sometimes better than 'Always'? : Intermittent assertions in proving program correctness, CACM, Vol. 21, No. 2, 1978.

10. A. Pnueli : The temporal semantics of concurrent programs, Lect. Notes in Comp. Sci. 70, Springer, 1979.

11. Z. Manna and A. Pnueli : The modal logic of programs, Lect. Notes in Comp. Sci. 71, ALP, 1979.

12. C. B. Schwind : Representing actions by state logic, Proc. of 3rd AISB/GI Conference, 1978.

13. J. McCarthy and P. J. Hayes : Some philosophical problems from the standpoint of artificial intelligence, Machine Intelligence 4, 1969.

14. M. Sato : On McCarthy's modal axiomatization of Knowledge, Proc. of the Ist IBM Symp. on MECS, The theory of programs and its surroundings, IBM, Japan, 1976.

15. R. Montague : Universal grammar, in Formal philosophy-selected papars of R. Montague, edited by R. H. Tomason, Yale Univ. Press, 1977.

16. R. Montague : The proper treatment of quantification in ordinary English, ibid.

17. T. M. W. Janssen and P. van Emde Boas : On the proper treatment of referencing, dereferencing and assignment, Lect. Notes in Comp. Sci. 52, Springer, 1977.

18. T. M. W. Janssen and P. van Emde Boas : The expressive power of intensional logic in the semantics of programming languages, Lect. Notes in Comp. Sci. 53, 1977.

19. C. A. R. Hoare : An axiomatic basis for computer programming, CACM, Vol. 12, No. 10, 1969.

20. R. W. Floyd : Assigning meanings to programs, Proc. Amer. Math. Soc. Symposia in Applied Mathematics, Vol. 19, 1967.

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

22. L. Henkin : A theory of propositional types, Fund. Math., 52, 1963.

23. E. W. Dijkstra : A discipline of programming, Prentice-Hall, 1976.

24. R. Yeh : Strong verification of programs, IEEE Transactions on SE, Vol. SE-1, No. 3, 1975.

25. J. McCarthy : A basis for a mathematical theory of computation, in P. Braffort and D. Hirschberg eds. : Computer programming and formal systems, 1970.

26. Z. Manna : Properties of programs and the first-order predicate calculus, JACM, Vol. 16, No. 2, 1969.

27. J. A. Robinson : A note on mechanizing higher order logic, Machine Intelligence 5, 1970.

28. G. Wrightson : A proof procedure for higher-order modal logic, 4th Workshop on Automated Deduction, 1979.