

TOWARDS A MATHEMATICAL THEORY OF PROGRAM SYNTHESIS

Masahiko Sato*

Department of Information Science
Faculty of Science, University of Tokyo
Hongo, Tokyo 113, Japan

This paper attempts to set a theoretical foundation for Program Synthesis. A programming language Prog is introduced together with its interpretation in the semantic domain D. A formal system QFT, which is an extension of the Heyting arithmetic, is then introduced in order to express specifications formally. Roughly stated, the main result is: Given a QFT-proof of the specification $\forall x \exists z A[x, z]$, it is possible to construct a program p such that $A[x, p[x]]$ is true in D for any input x. Moreover, a formal proof of $\forall x A[x, p[x]]$ can also be obtained. Since any program p in Prog always terminates, the above result establishes the total correctness of the synthesized program at the same time. Finally, it is shown that any program p in Prog can be faithfully simulated by means of LISP.

0. INTRODUCTION

Before we start talking about *program synthesis*, we wish to clarify our standpoint as to what we mean by the terms such as *program*, *specification* and *verification*. A *program* is a well-formed expression in some formal language (i.e., a *programming language*). We write a program in order to solve a certain problem. We need a theory to specify the problem precisely. (See [Burstall and Goguen 1977]. They give an example of matrix inversion. I.e., we need a theory of matrices to state the problem of matrix inversion.) As far as program synthesis is concerned, we must further have a formal language (i.e., a *specification language*) in which the above theory can be formalized. Then, in the specification language, we can *specify* the problem formally by a formula $A[x, z]$ which describes an input-output relation between the input variable x and the output variable z. We say that a program P *realizes* the specification $A[x, z]$ if, under a certain interpretation, $A[x, P[x]]$ is true for any input x, where $P[x]$ denotes the result of executing P for the input x. Here we remark that we are implicitly assuming that our specification language essentially contains the programming language in which the program P is written. If we furthermore wish to formally *verify* that P realizes the specification $A[x, z]$, we need a formal theory such that the formula $\forall x A[x, P[x]]$ is provable in it. Thus this theory must contain the theory which we required for the specification of the problem. We call the former theory *verification theory* and the latter theory *specification theory*. Now we can formulate the problem of program synthesis as follows.

Program Synthesis Problem (PSP): Find a procedure which produces, from a given specification $A[x, z]$, a program P which realizes the specification.

Even though at a rather meta-level, we need a theory to specify PSP. As the specification theory of PSP, we adopt the theory QFT which is a quantificational extension of the theory FT of primitive recursive functionals of finite type. (Theory of primitive recursive functionals of finite type was initiated by [Gödel 1958], as a device to prove the consistency of the Peano arithmetic. Our systems QFT and FT are very similar to the corresponding systems due to [Schütte 1977].) The reasons of our adoption of QFT and FT are as follows.

- (1) Every provably recursive function (in the sense of [Takeuti 1975] p.123) is expressible in FT, so that a large class of interesting algorithms can be dealt with.
- (2) QFT contains the Heyting arithmetic HA as a subsystem, so that we can discuss the synthesis of programs which compute number theoretic functions.
- (3) Every functional of type $\sigma \rightarrow \tau$ in FT may be regarded as a program which computes a functional of type τ from a given functional of type σ . Thus we can discuss the synthesis of a program which yields another program from a given program, in case both σ and τ are not of type 0.
- (4) Important proof-theoretic results concerning FT and QFT can be used as theoretical foundations for program synthesis.
- (5) The computation in FT can be easily simulated by LISP.

We will solve PSP under the following framework:

- (a) A program is a (closed) term in FT.
- (b) The specification theory is QFT.
- (c) The verification theory is QFT.

Suppose that the specification $A[x, z]$ of a problem is given. Then either by a theorem prover

* This work was supported in part by the Sakkokai Foundation.

or by a man-machine interactive proof checker, we try to construct a QFT-proof π of the formula $\forall x \exists z A[x, z]$. (This is the most challenging part of our procedure, since no decision procedure exists for the provability of a QFT-formula. Other parts of our procedure are completely mechanical.) If we succeed in constructing such a π , then from π we can construct, primitive recursively, a program P such that $A[x, P[x]]$ is provable in QFT (Theorem 9). Furthermore we can formally verify that P realizes the specification, since $\forall x A[x, P[x]]$ is provable in QFT. We note that, according to Theorem 1, the program P always terminates for any input x . Hence, the total correctness of the program will be established at the same time.

We think that it is fundamental for any program synthesis approach, which may be called mathematically well-founded, to satisfy the following conditions.

1. The semantics of the programming language is precisely specified.
2. The specification theory is precisely specified.
3. The verification theory is precisely specified.

However, it seems to us that many of the existing approaches to program synthesis are too naive to satisfy any one of the above criteria. In contrast, our approach to be described below satisfies all of the above criteria, and hence is mathematically well-founded.

1. FUNCTIONALS OF FINITE TYPE

1.1 Types

Definition of types:

1. The symbol 0 is a type.
 2. If σ and τ are types then $[\sigma \rightarrow \tau]$ is a type.
- For brevity we write $\sigma_1 \rightarrow \sigma_2 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$ instead of $[\sigma_1 \rightarrow [\sigma_2 \rightarrow \dots \rightarrow [\sigma_n \rightarrow \tau] \dots]]$. The set of all the types will be denoted by $Type$.

1.2 Terms and programs

The terms of FT will be constructed from the following language LFT:

1. Denumerably infinite set FV^σ of free variables for each type σ : $FV^\sigma = \{a_1^\sigma, a_2^\sigma, a_3^\sigma, \dots\}$. We put $FV = \cup \{FV^\sigma \mid \sigma \in Type\}$.
2. Denumerably infinite set BV^σ of bound variables for each type σ : $BV^\sigma = \{x_1^\sigma, x_2^\sigma, x_3^\sigma, \dots\}$. We put $BV = \cup \{BV^\sigma \mid \sigma \in Type\}$.
3. Constants: 0 (zero), S (successor) and J_σ (iterator) for each type σ .
4. Lambda abstraction symbol: λ .
5. Parentheses: $($ and $)$.

As auxiliary symbols, we have denumerably infinitely many nominal symbols: $*1, *2, *3, \dots$. By an n -place (LFT) nominal form we mean a finite string of symbols which contains, in addition to the symbols of LFT, at most the nominal symbols $*1, *2, \dots, *n$, where n is the decimal numeral which re-

presents the natural number n . If A is an n -place nominal form and $r_1, \dots, r_n$ are strings of symbols, then $A[r_1, \dots, r_n]$ denotes the result of replacing every occurrence of $*i$ in A by r_i ($1 \leq i \leq n$). We will use *italic* letters to denote nominal forms.

Definition of the set Tm of terms, type: $Tm \rightarrow Type$

and $bvar: Tm \rightarrow 2^{BV}$:

1. $a \in FV \Rightarrow a \in Tm, type[a] = \sigma, bvar[a] = \emptyset$.
2. $0 \in Tm, type[0] = 0, bvar[0] = \emptyset$.
3. $S \in Tm, type[S] = 0 \rightarrow 0, bvar[S] = \emptyset$.
4. $J_\sigma \in Tm, type[J_\sigma] = 0 \rightarrow [\sigma \rightarrow \sigma] \rightarrow \sigma \rightarrow \sigma, bvar[J_\sigma] = \emptyset$.
5. $r, s \in Tm, type[r] = \sigma \rightarrow \tau, type[s] = \sigma, bvar[r] \cap bvar[s] = \emptyset \Rightarrow (r \ s) \in Tm, type[(r \ s)] = \tau, bvar[(r \ s)] = bvar[r] \cup bvar[s]$.
6. $r[a] \in Tm, type[r[a]] = \tau, a \in FV^\sigma, x \in BV^\sigma - bvar[r[a]] \Rightarrow \lambda x r[x] \in Tm, type[\lambda x r[x]] = \sigma \rightarrow \tau, bvar[\lambda x r[x]] = bvar[r[a]] \cup \{x\}$.

Tm is the least set which satisfies the above conditions. We put $Tm^\sigma = \{t \in Tm \mid type[t] = \sigma\}$. An element in Tm is called a *term of type σ* . Two terms r and s are *congruent* if they are constructed exactly in the same manner, except that possibly different bound variables are selected in the applications of 6 above. We will henceforth identify congruent terms. This identification has the effect of making bound variables really dummy. A *program* is a term containing no free variables. We write $Prog$ for the set of programs, and we put $Prog^\sigma = \{t \in Prog \mid type[t] = \sigma\}$. We use r, s, t etc. as syntactic symbols denoting terms or programs. For brevity we write $r_1 r_2 r_3 \dots r_n$ instead of $((\dots (r_1 r_2) r_3) \dots) r_n$.

Definition of a binary relation red on $Prog$:

1. $(\lambda x r[x] \ s) \ red \ r[s]$.
2. $J_\sigma \ 0 \ st \ red \ t$.
3. $J_\sigma^\sigma (Sr) \ st \ red \ s(J_\sigma \ rst)$.
4. $r^\sigma \ red \ r' \Rightarrow (r \ s)^\sigma \ red \ (r' \ s)$.
5. $s \ red \ s' \Rightarrow (r \ s) \ red \ (r \ s')$.

We let red^* be the reflexive and transitive closure of red . A program p is *normal* if $p \ red \ q$ for no q .

Definition of the numerals N_n (n : a natural number):

1. $N_0 = 0$.
2. $N_{n+1} = (S \ N_n)$.

The following three lemmas can be proved by a standard technique. (See, e.g., [Schütte 1977], [Troelstra 1973], [Howard 1970].)

Lemma 1. A program p of type 0 is normal if and only if $p = N_n$ for some natural number n .

Lemma 2. If $p \ red^* q$, $p \ red^* q'$, and both q and q' are normal, then $q = p'$.

Lemma 3. For any program p there exists a normal program q such that $p \ red^* q$.

Putting Lemma 2 and Lemma 3 together, we have:

Theorem 1. For any program p there uniquely exists a normal program q such that $p \ red^* q$. The program q in the above theorem is called the *normal form* of p , and is denoted by $[p]$. We write D for the set of programs in normal form,

and we put $D^0 = \{p \in D \mid \text{type}[p] = \sigma\}$.

1.3 Interpretation of terms

We can use the function $\llbracket \cdot \rrbracket : \text{Prog} \rightarrow D$ which sends a program p to its normal form $\llbracket p \rrbracket$, as a semantic function. For instance, a program p of type $\sigma \rightarrow \tau$ may be considered as denoting a functional $\hat{p} : D^0 \rightarrow D^0$ such that $\hat{p}[s] = \llbracket (p \ s) \rrbracket$ for any $s \in D^0$. Fur-

thermore, we remark that, by Lemma 1, D^0 is isomorphic with the set $\mathbb{N}$ of natural numbers. With this interpretation in mind, we define the formal system FT of primitive recursive functionals of finite type as follows.

2. THE FORMAL SYSTEM FT

2.1 Definition of the system FT

Definition of the set Form of FT-formulas:

1. $s, t \in \text{TM}^0 \Rightarrow s = t \in \text{Form}$.
 2. $A, B \in \text{Form} \Rightarrow (A \supset B), (A \wedge B), (A \vee B) \in \text{Form}$.
- We use Γ, Δ etc. to denote finite (possibly empty) sequences of formulas. If Γ is the sequence $'A_1, \dots, A_n'$ then $\Gamma \supset B$ is the abbreviation of $A_1 \supset \dots \supset A_n \supset B$ which, in turn, is the abbreviation of $(A_1 \supset (\dots \supset (A_n \supset B) \dots))$. If $\Gamma = 'A_1, \dots, A_n'$ and $\Delta = 'B_1, \dots, B_m'$ then $\Gamma \subseteq \Delta$ means that the set $\{A_1, \dots, A_n\}$ is included in the set $\{B_1, \dots, B_m\}$. We define

the connective $\neg$ by: $\neg A = A \supset 0 = 0$. A formula containing no free variable is called a *closed* formula. A formula which only contains equations between terms of type 0 is said to be a *basic formula*.

Definition of the provability relation $\vdash$ in FT:
Axioms:

- (λ) $\vdash (\lambda x r[x] \ s) = r[s]$.
- (J 0) $\vdash J_0 st = t$.
- (J $^0_\sigma$) $\vdash J^0_\sigma (Sr) st = s(J_\sigma rst)$.
- (=) $\vdash s = t \supset F[s] \supset F[t]$.

Inference rules:

- (Str) $\vdash \Gamma \supset B, \Gamma \subseteq \Delta \Rightarrow \vdash \Delta \supset B$.
- (Cut) $\vdash \Gamma \supset A, \vdash A \supset B \Rightarrow \vdash \Gamma \supset B$.
- (CI) $\vdash F[0], \vdash F[a] \supset F[sa], a \in \text{FV}^0, a \notin F, t \in \text{TM}^0 \Rightarrow \vdash F[t]$.
- ($\wedge$ l) $\vdash A \supset B \supset C \Rightarrow \vdash (A \wedge B) \supset C$.
- ($\wedge$ r) $\vdash \Gamma \supset A, \vdash \Gamma \supset B \Rightarrow \vdash \Gamma \supset (A \wedge B)$.
- ($\vee$ l) $\vdash A \supset C, \vdash B \supset C \Rightarrow \vdash (A \vee B) \supset C$.
- ($\vee$ r) $\vdash \Gamma \supset A_i \Rightarrow \vdash \Gamma \supset (A_1 \vee A_2)$ ($i=1,2$).

In the above definition, $a \notin F$ means that the free variable a does not occur in the nominal form F . A formula A is said to be *provable* (in FT) if $\vdash A$.

2.2 Consistency and weak completeness

Definition of *truth* and *false* for a closed formula:

1. A closed formula $s=t$ is *true* iff $\llbracket s \rrbracket = \llbracket t \rrbracket$.
2. A closed formula $A \supset B$ is *true* iff A is *false* or B is *true*.
3. A closed formula $A \wedge B$ is *true* iff A is *true* and B is *true*.
4. A closed formula $A \vee B$ is *true* iff A is *true* or

B is *true*.

5. A closed formula A is *false* iff A is not *true*. We write $\models A$ if A is *true*. We then have the following consistency theorem. (See [Schütte 1977], Theorem 17.2.)

Theorem 2. $\vdash A, A: \text{closed} \Rightarrow \models A$.

Although not all true closed formulas are provable in FT (example: $\neg J_0(S0)S=S$), we have the following weak completeness theorem.

Theorem 3. $\models A, A: \text{basic} \Rightarrow \vdash A$.

Putting Theorem 2 and Theorem 3 together, we have:

Theorem 4. A closed basic formula is provable in FT if and only if it is true.

3. THE FORMAL SYSTEM QFT

3.1 Definition of QFT

Definition of the set Form of QFT-formulas and

- bvar: $\text{Form} \rightarrow 2^{\text{BV}}$:
1. $s, t \in \text{TM}^0 \Rightarrow s = t \in \text{Form}$, $\text{bvar}[s=t] = \text{bvar}[s] \cup \text{bvar}[t]$.
2. $A, B \in \text{Form} \Rightarrow (A \supset B), (A \wedge B), (A \vee B) \in \text{Form}$, $\text{bvar}[(A \supset B)] = \text{bvar}[(A \wedge B)] = \text{bvar}[(A \vee B)] = \text{bvar}[A] \cup \text{bvar}[B]$.
3. $A[a] \in \text{Form}, a \in \text{FV}^0, x \in \text{BV}^0 - \text{bvar}[A[a]] \Rightarrow \forall x A[x], \exists x A[x] \in \text{Form}$, $\text{bvar}[\forall x A[x]] = \text{bvar}[\exists x A[x]] = \text{bvar}[A[a]] \cup \{x\}$.

Two formulas are *congruent* if they are constructed exactly in the same manner, except that possibly different bound variables are selected in the applications of 3 above. We will henceforth identify congruent formulas. A formula is *quantifier free* if it contains no occurrence of $\forall$ or $\exists$. A quantifier free QFT-formula is just a basic FT-formula. We define the negation $\neg A$ of a formula A by $\neg A = A \supset 0 = 0$.

Notational conventions: Let $s = 's_1, \dots, s_n'$ be a sequence of terms $s_1, \dots, s_n$ and let $t = 't_1, \dots, t_m'$ be a sequence of terms $t_1, \dots, t_m$, where $m, n \geq 0$. Suppose that $\text{type}[s_i] = \sigma_i, 1 \leq i \leq n$ and $\text{type}[t_j] = \tau_j, 1 \leq j \leq m$. Then we let $\{s \ t\}$ denote the sequence $'s_1, t_1, \dots, s_n, t_m'$. We use r, s, t etc. to denote sequence of terms. Similarly, we use a, b, c etc. to denote sequences of free variables, and x, y, z etc. to denote sequences of bound variables. If $x = 'x_1, \dots, x_n'$, $t = 't_1, \dots, t_n'$ and A is an n -place nominal form, then $\forall x A[x]$, $\exists x A[x]$ and $A[t]$ are the abbreviations of $\forall x_1 \dots \forall x_n A[x_1, \dots, x_n]$, $\exists x_1 \dots \exists x_n A[x_1, \dots, x_n]$ and $A[t_1, \dots, t_n]$, respectively.

Definition of the provability relation $\vdash$ in QFT:

Axioms:

- (FT) $\vdash A$ (in FT), $A: \text{basic} \Rightarrow \vdash \neg A$.
- ($\forall$ l) $\vdash \forall x A[x] \supset A[t]$.
- ($\exists$ r) $\vdash A[t] \supset \exists x A[x]$.
- (IP' $_0$) $A: \text{quantifier free} \Rightarrow \vdash (\forall x A[x] \supset \exists y B[y]) \supset \exists y (\forall x A[x] \supset B[y])$.
- (M') $A: \text{quantifier free} \Rightarrow \vdash \neg \exists x A[x] \supset \exists x A[x]$.
- (AC) $\vdash \forall x \exists y A[x, y] \supset \exists z \forall x A[x, z]$.

Inference rules

(Str), (Cut), (CI), ($\wedge$ l), ($\wedge$ r), ($\vee$ l) ($\vee$ r): same

as in FT for QFT-formulas.

($\forall$ r) $\vdash \Gamma \supset A[a], a \notin A \Rightarrow \vdash \Gamma \supset \forall x A[x]$.

($\exists$ l) $\vdash A[a] \supset B, a \notin A \Rightarrow \vdash \exists x A[x] \supset B$.

In the above definition, $a \notin A$ means that none of the free variables in the sequence a occurs in the nominal form A . Also in the above definition, IP, M and AC stand for 'independence of premiss', 'Markov's principle' and 'axiom of choice', respectively. Two formulas A and B are *equivalent* in QFT if both $A \supset B$ and $B \supset A$ are provable in QFT.

3.2 Interpreting formulas

A QFT-formula F is an *interpreting formula* if it is of the form $\exists x \forall y F_0[x, y]$, and F_0 is quantifier free. (We use the subscript 0 to denote a quantifier-free nominal form.)

Definition of Gödel translation: With each QFT-formula F , we associate an interpreting formula F^* (Gödel translation of F) as follows.

1. F : quantifier free $\Rightarrow F^* = F$.

Let $A^* = \exists x_1 \forall y_1 A_0[x_1, y_1]$ and $B^* = \exists x_2 \forall y_2 B_0[x_2, y_2]$.

2. $F = A \supset B \Rightarrow F^* = \exists x_2 \exists y_1 \forall x_1 \forall y_2 (A_0[x_1, y_1] \supset B_0[x_2, y_2])$.

3. $F = A \wedge B \Rightarrow F^* = \exists x_1 \exists x_2 \forall y_1 \forall y_2 (A_0[x_1, y_1] \wedge B_0[x_2, y_2])$.

4. $F = A \vee B \Rightarrow F^* = \exists z \exists x_1 \exists x_2 \forall y_1 \forall y_2 (z = 0 \supset A_0[x_1, y_1] \wedge z = 0 \supset B_0[x_2, y_2])$.

Let $(A[a])^* = \exists x_1 \forall y_1 A_0[x_1, y_1, a]$

5. $F = \forall x A[x] \Rightarrow F^* = \exists x_1 \forall x \forall y_1 A_0[\tilde{x}_1, x, y_1, x]$.

6. $F = \exists x A[x] \Rightarrow F^* = \exists x_1 \exists x \forall y_1 A_0[x_1, y_1, x]$.

The following theorem can be verified without difficulty. (AC) plays a role in the proof.

Theorem 5. For any QFT-formula F , F and F^* are equivalent in QFT.

3.3 Reduction of QFT to FT

Let $F = \exists x \forall y F_0[x, y]$ be an interpreting formula. A quantifier free formula $F_0[t, a]$ is called a *reduction* of F , if a is a sequence of free variables such that $a \notin F_0$, and if t is a sequence of terms whose free variables, if any, always occur in F_0 . Note that $F_0[t, a]$ is a basic FT-formula. Also, note that t is closed if F_0 contains no free variable. We then have the following important theorem due originally to [Gödel 1958]. (Gödel gives the reduction of the Heyting arithmetic HA to FT. Reduction of a system which admits quantifications over functionals of any type to FT was first carried out by [Yasugi 1963]. For a correction to [Yasugi 1963], see [Troelstra 1972].)

Theorem 6. If a QFT-proof of an interpreting formula $F = \exists x \forall y F_0[x, y]$ is given, then we can construct, primitive recursively, a reduction $F_G = F_0[t, a]$ of F and its proof in FT.

We call above F_G the Gödel reduction of F . By Theorem 2 and Theorem 6, we have the following consistency theorem for QFT.

Theorem 7. The system QFT is syntactically consistent, i.e., there exists a QFT-formula A which is not provable in QFT.

3.4 Truth definition for QFT-formulas

Definition of *truth* and *false* for a closed QFT-formula:

1. $A = s = t \Rightarrow A$ is true iff $[s] = [t]$.

2. $A = B \supset C \Rightarrow A$ is true iff B is false or C is true.

3. $A = B \wedge C \Rightarrow A$ is true iff B is true and C is true.

4. $A = B \vee C \Rightarrow A$ is true iff B is true or C is true.

5. $A = \forall x B[x], x \in BV^0 \Rightarrow A$ is true iff $B[t]$ is true for all $t \in D^0$.

6. $A = \exists x B[x], x \in BV^0 \Rightarrow A$ is true iff $B[t]$ is true for some $t \in D^0$.

7. A is false iff A is true.

In contrast to the constructive definition of truth for FT-formulas, the above truth definition is nonconstructive because of 5 and 6 above. A true QFT-formula will be referred to as *classically true* when we wish to emphasize this nonconstructive character of truth definition. Anyway, we remark that the above definition gives a (at least classically) natural interpretation of QFT-formulas in the domain D .

We now extend the truth definition to any QFT-formulas as follows. An *assignment* is a type preserving function $\phi: FV \rightarrow D$. Let ϕ be an assignment, and A be any QFT-formula. Let $a = 'a_1, \dots, a_n'$ be the stock of all free variables occurring in A (a is empty if A is closed). Then A is of the form $A[a_1, \dots, a_n]$. Now, A is *true under* ϕ (notation: $\phi \models A$) if the closed formula $A[\phi[a_1], \dots, \phi[a_n]]$ is true. A is *valid* ($\models A$) if $\phi \models A$ for any assignment ϕ . A closed formula is valid iff it is true.

A tricky point about the system QFT is that the axiom schema (AC) is not always valid. For by Problem 9-(b) in Chapter 8 of [Shoenfield 1967],

there is a recursive function $f: D^0 \rightarrow D^0$ such that (i): $f \neq t$ for all $r \in D^{0+0}$, (See §1.3 for the definition of f . $f \neq t$ means that two functions are extensionally distinct, i.e., there exists some $s \in D^0$ such that $f[s] \neq t[s]$.) and (ii): there is a QFT-formula $A[a, b]$ such that $\phi \models A[a, b]$ iff $\phi[b] = f[\phi[a]]$. Hence the following instance of (AC) is not valid: $\forall x \exists y A[x, y] \supset \exists z \forall x A[x, z]$. (See also II-§5-2.5.3-(iv) of [Troelstra 1973].) Then, by Theorem 8 below, we see that the Gödel translation $F \rightarrow F^*$ does not always preserve the classical truth value.

The above observation suggests us to define the following variants of QFT: We define the system QFT₀ as the system which results from QFT by deleting the axiom (AC). We obtain the classical system CQFT₀ by adding the following axiom ($\neg$) to QFT₀:

($\neg$) $\neg \neg A \supset A$.

We have the following semantical consistency theorem.

Theorem 8. 1) QFT₀ $\vdash A \Rightarrow$ CQFT₀ $\vdash A \Rightarrow \models A$.
2) QFT₀ $\vdash \neg A \Rightarrow \models \neg A^*$.

4. PROGRAM SYNTHESIS

4.1. The problem of program synthesis

Definition of a specification: A *specification* is a triple $\Sigma = \langle A; 'b_1, \dots, b_n'; 'a_1, \dots, a_n' \rangle$ where

A is an $m+n$ -place nominal form such that $A[b, a] = A[b_1, \dots, b_n, a_1, \dots, a_m]$ is a formula containing at most b and a free. We put

$D_I(\Sigma) = D_{\tau_1}^1 \times \dots \times D_{\tau_m}^m$, where $\sigma_1 = \text{type}[a_1]$,

$D_O(\Sigma) = D_{\tau_1}^1 \times \dots \times D_{\tau_n}^n$, where $\tau_j = \text{type}[b_j]$ and

$R(\Sigma) = \{(s, t) \in D_I(\Sigma) \times D_O(\Sigma) \mid \models A[t, s]\}$.

We call $D_I(\Sigma)$ the *input domain* of Σ , $D_O(\Sigma)$ the *output domain* of Σ and $R(\Sigma)$ the *input-output relation* of Σ . An element s in $D_I(\Sigma)$ is called an *input*. We identify $s = (s_1, \dots, s_m) \in D_I(\Sigma)$ with the sequence $'s_1, \dots, s_m'$. Two specifications Σ and Σ' are *equivalent* if $R(\Sigma) = R(\Sigma')$. A sequence of programs will simply be called a *program*.

Definition of realization: A program p *realizes* the specification Σ if $A[ps, s]$ is true for any input $s \in D_I(\Sigma)$.

A specification Σ then determines the following.

Program synthesis Problem PSP(Σ): "Construct a program p which realizes the specification Σ ."

In order that PSP(Σ) may have a solution, it is necessary that the formula $B = \exists y \forall x A[yx, x]$ is true. (Because if a program p realizes Σ , then $\models B$.) We note that B is true iff there exists a program $p = 'p_1, \dots, p_n'$ such that $\{(s, \beta[s]) \in D_I \times D_O \mid s \in D_I\} \subseteq$

$R(\Sigma)$, where $\beta[s] = (\beta_1[s], \dots, \beta_n[s])$ and $\beta_i[s] = [(p_i, s)]$. Roughly speaking, B is true iff $R(\Sigma)$ contains the graph of a program. We also remark that the formula B semantically entails the formula $C = \forall x \exists z A[z, x]$. The formulas B and C are equivalent in QFT, but, as we have seen in §3.4, they are not necessarily semantically equivalent. By a *prenex formula* we mean a formula of the form $Q_1 x_1 \dots Q_n x_n A_0[x_1, \dots, x_n]$, where $Q_1 x_1$ is either

$\forall x_1$ or $\exists x_1$. Since CQFT_0 is a classical system, any formula A can be transformed into a prenex formula B , which we call the *prenex normal form* of A , such that A and B are equivalent in CQFT_0 . We have the following easily verifiable lemma.

Lemma 4. If A is a prenex formula, then $\text{QFT}_0 \vdash A^* \supset A$.

We are now in a position to prove the following Main Theorem:

Theorem 9. Let $\Sigma = \langle A; b; a \rangle$ be a specification, and let $B[b, a]$ be the prenex normal form of $A[b, a]$. Then if a QFT-proof of the formula $\forall x \exists y B[y, x]$ is given, we can construct, primitive recursively, a program p which realizes the specification Σ . Moreover, we can formally verify that p realizes Σ , i.e., the formula $\forall x A[px, x]$ is provable in CQFT_0 .

Proof. Define a nominal form B^* by $B^*[b, a] = (B[b, a])^*$. Then $B^*[b, a]$ is of the form $\exists u \forall v B_0[b, a, u, v]$. Let $C = \forall x \exists y B[y, x]$. Then $C^* = (\forall x \exists y B^*[y, x])^* =$

$\exists z \exists w \forall x \forall v B_0[zx, x, wx, v]$. Then by Theorem 6, we have $\text{FT} \vdash (C^*)_G$, where $(C^*)_G$ is of the form $B_0[pa, a, qa, c]$ (p and q are sequences of closed terms, i.e., programs, since C^* is closed, and a, c are sequences of free variables). Then we have:

$\text{FT} \vdash B_0[pa, a, qa, c]$
 $\Rightarrow \text{QFT}_0 \vdash \exists w \forall x \forall v B_0[px, x, wx, v]$ (by $(\forall r), (\exists r), (MP)$)
 $\Rightarrow \text{QFT}_0 \vdash \forall x \exists u \forall v B_0[px, x, u, v]$ (by Lemma 4)
 $\Rightarrow \text{QFT}_0 \vdash \exists u \forall v B_0[pa, a, u, v]$ (by $(\forall 1), (MP)$)
 $\Rightarrow \text{QFT}_0 \vdash B[pa, a]$ (by Lemma 4)
 $\Rightarrow \text{CQFT}_0 \vdash A[pa, a]$ (by assumption)
 $\Rightarrow \text{CQFT}_0 \vdash \forall x A[px, x]$ (by $(\forall r)$).

We have thus established the second part of the theorem. That p realizes Σ now follows by Theorem 8-(1).

Remark. 1. If A is quantifier free, then B is identical with A . We note that this category already contains many working examples such as quotient-remainder, GCD etc. For more examples see [Takahashi 1977].

2. Note that we have solved the problem by changing the given specification Σ to its equivalent variant $\Sigma' = \langle B; b; a \rangle$, providing the problem does not fall in the category mentioned above. (A and B are equivalent in CQFT_0 !) That this modification is necessary can be seen as follows.

Let $Ac = \forall x \exists y F[x, y] \supset \exists z \forall x F[x, zx]$ be a false instance of (AC). (See §3.4.) Let Σ be the specification $\langle A; 'b'; 'a' \rangle$ where $A = Ac \wedge 1 = 0$ and $\text{type}[a] = \text{type}[b] = 0$ so that $A[b, a] = Ac \wedge b = 0$. Then $\Sigma' = \langle B; 'b'; 'a' \rangle$, where $B[b, a]$ is the prenex normal form of $A[b, a]$. Now, if we feed the formula $\forall x \exists y A[y, x]$ to a QFT-theorem-prover, then since it is provable in QFT_0 , we will have a program $p = \lambda x 0$ whose graph is $D \times \{0\}$. But $R(\Sigma)$ is empty since $Ac \wedge b = 0$ is false under any assignment ϕ . Thus our program p does not realize the specification Σ . (Caution: That a false formula is provable in QFT does not imply its inconsistency. See Theorem 8-(2).) We have avoided such awkwardness by shifting from Σ to its equivalent variant Σ' . In this case the formula $\forall x \exists y B[y, x]$ is simply unprovable in QFT.

3. There are several interesting relations between the inference rules employed in the proof of $B[b, a]$ and the structure of the synthesized programs, of which we cannot go into details because of the limitation of space. Roughly speaking, (MP), $[(\forall \rightarrow), (\rightarrow \wedge), (\text{Str})]$ and (CI) correspond to composition of programs, conditional (*if_then_else*) and primitive recursion, respectively. In particular, a structured proof yields a structured program.

4.2 Simulation by LISP

We have seen that the semantic function $[] : \text{Prog} \rightarrow D$ is recursive (Theorem 1). In order to simulate this function by a LISP S-function, we proceed as follows. Let Sexp denote the set of LISP S-expressions.

Definition of $\psi : \text{Prog} \rightarrow \text{Sexp}$:

$\psi[0] = 0$, $\psi[S] = S$, $\psi[J_\sigma] = J$, $\psi[x \bar{1}^\sigma] = x \bar{1}$

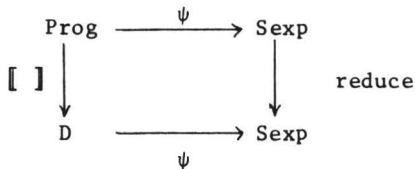
$\psi[(r\ s)] = (\psi[r]\ \psi[s])$, $\psi[\lambda x r[x]] = (L\ \psi[x]\ \psi[r[x]])$.
Definition of the S-function reduce:

```

reduce[x] =
[normal[x] → x;
atom[caar[x]] → [eq[caar[x]; L] → reduce[lconv[x];
                    T → subreduce[x] ];
atom[caaar[x]] →
[eq[caaar[x]; J] → [atom[cadaar[x] → reduce[cadr[x]];
                    atom[caadaar[x]] → [eq[caadaar[x]; S] →
reduce[list[cadar[x]; list[list[list[J; caadaar[x]]; cadr[x]]];
                    cadr[x] ]]]];
                    T → subreduce[x] ];
                    T → subreduce[x] ];
                    T → subreduce[x] ];
subreduce[x] = reduce[list[reduce[car[x]]; reduce[cadr[x]]]]
normal[x] =
[atom[x] → T;
atom[car[x]] → subnormal[x];
atom[caar[x]] → [eq[caar[x]; L] → NIL; T → subnormal[x]];
atom[caaar[x]] → [eq[caaar[x]; J] →
[atom[cadaar[x]] → [eq[cadaar[x]; 0] → NIL;
                    T → subnormal[x] ];
                    atom[caadaar[x]] → [eq[caadaar[x]; S] → NIL;
                    T → subnormal[x] ];
                    T → subnormal[x] ];
                    T → subnormal[x] ];
subnormal[x] = [normal[car[x]] → normal[cadr[x]]; T → NIL]
lconv[x] = lconv1[caddar[x]; cadr[x]; cadr[x]]
lconv1[x; v; y] = lcnv[x; NIL; v; y]
lcnv[x; bv; v; y] =
[atom[x] → [eq[x; v] → [memq[v; bv] → x; T → y]; T → x];
atom[car[x]] → [eq[car[x]; L] → list[L; cadr[x]; lcnv[caddar[x];
                    append[cons[cadr[x]; NIL]; bv; v; y ]];
                    T → list[lcnv[car[x]; bv; v; y];
                    lcnv[cadr[x]; bv; v; y] ] ];
                    T → list[lcnv[car[x]; bv; v; y]; lcnv[cadr[x]; bv; v; y] ] ]

```

Theorem 10. The diagram below commutes:



Thus we have seen that the function reduce simulates the semantic function $[]$. If ψ were a one-to-one function, then we could compute $[]$ by the equation $[p] = (\psi^{-1} \cdot \text{reduce} \cdot \psi)[p]$. But ψ is not one-to-one since $\psi[J] = J$ for any σ Type. However, we note that $\psi \times \text{type} : \text{Prog} \rightarrow \text{Sexp} \times \text{Type}$ such that $(\psi \times \text{type})[p] = (\psi[p], \text{type}[p])$ is one-to-one. Hence we can compute $[]$ by the scheme $[p] = (\psi \times \text{type})^{-1}[\text{reduce}[\psi[p]], \text{type}[p]]$. (Note that $[]$ preserves type.)

4.3 Concluding Remarks

We believe that we have successfully set a theoretical foundation for program synthesis. It is not difficult to implement our procedure, and we wish to encourage the interested readers to actually implement it. In fact, at a suggestion of the author, [Goto 1978] has implemented a system

of program synthesis. However, his system differs at an essential point from the one described above, and, at present, no proof exists which guarantees the correctness of the program synthesized by Goto's system. The possibility of using Gödel interpretation as a tool for program synthesis has also been suggested by [Kreisel 1977]. [Takasu 1978] also studies the application of Gödel interpretation to program synthesis using the formalism of Gentzen's sequential calculus. Since the basic data type of Prog is natural number, it may appear that such data types as list, stack etc. cannot be dealt with in our formalism. However it is not the case, and we can treat these data types as well by treating them as abstract data types. We wish to discuss this point further in a paper to be published later.

ACKNOWLEDGMENTS.

The author wishes to thank Professor Satoru Takasu of Kyoto University for many stimulating discussions with the author. The author wishes to thank Mr. Hayao Nakahara for correcting errors in the original definition of the function reduce.

REFERENCES

- [Burstall, R.M. and Goguen, J.A. 1977] *Putting theories together to make specifications*, Proc. IJCAI-77, MIT, Cambridge, Mass., 1045-1058.
- [Gödel, K. 1958] *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes*, Dialectica, 12, 280-287.
- [Goto, S. 1978] *Program synthesis through Gödel's interpretation*, Proc. International Conference on Mathematical Studies of Information Processing, 287-306, Kyoto.
- [Howard, W.A. 1970] *Assignment of ordinals to terms of primitive recursive functionals of finite type*, Intuitionism and Proof Theory, 443-458, North-Holland.
- [Kreisel, G. 1977] *Some uses of proof theory for finding computer programs*, Colloque International des Logique, 123-134, Paris.
- [Shoenfield, J.R. 1967] *Mathematical Logic*, Addison Wesley.
- [Shütte, K. 1977] *Proof Theory*, Springer.
- [Takahashi, M. 1977] *A foundation of finite mathematics*, Publ. RIMS, Kyoto Univ., 12, 577-708.
- [Takasu, S. 1978] *Proofs and programs*, Proc. 3rd IBM symposium on Mathematical Foundations of Computer Science, IBM Japan.
- [Takeuti, G. 1975] *Proof Theory*, North-Holland.
- [Troelstra, A.S. 1972] *Review of [Yasugi 1963]*, J. of Symbolic Logic, 37, 404.
- [Troelstra, A.S. 1973] *Mathematical Investigations of intuitionistic arithmetic and analysis*, Lecture Notes in Mathematics 344, Springer.
- [Yasugi, M. 1963] *Intuitionistic analysis and Gödel's interpretation*, J. Math. Soc. Japan, 15, 101-112.