Realizability for Constructive Theory of Functions and Classes and Its Application to Program Synthesis

Makoto Tatsuta

Department of Mathematics
Kyoto University
Kyoto 606-8502, JAPAN
e-mail: tatsuta@kusm.kyoto-u.ac.jp

Abstract

This paper gives a **q***-realizability interpretation for Feferman's constructive theory* $\mathbf{T}_0$ *of functions and classes by using a set completion program and without doubling variables, and proves its soundness. This result solves an open problem proposed by Feferman in 1979. Moreover by using this interpretation we can prove a program extraction theorem for* $\mathbf{T}_0$*, which enables us to use constructive sets of* $\mathbf{T}_0$ *for program synthesis.*

1 Introduction

Feferman proposed the theory $\mathbf{T}_0$ of functions and classes in 1979 [3]. Beside the area of mathematical logic, the theory $\mathbf{T}_0$ has been studied in the area of computer science [6, 11], for the following features it enjoys.

(1) Feferman's theory $\mathbf{T}_0$ gives a logical system in which we can formalize properties of programs and develop formal theories for verification, synthesis and transformation of programs. By including λ-terms as a subsystem, $\mathbf{T}_0$ can formalize properties of programs. Moreover $\mathbf{T}_0$ is based on a logic of partial terms, which can formalize termination properties of programs. $\mathbf{T}_0$ has an explicit formula $a{\downarrow}$ for saying that the term a has a value, that is, computation of the term a is terminating, and $\mathbf{T}_0$ can also treat terms which may not have a value. In this sense, the type-free framework of $\mathbf{T}_0$ is more flexible for formalizing programs than usual type theoretic frameworks.

(2) The theory has constructive sets called classes, which can represent expressively and powerfully properties such as specifications. Sets are introduced by elementary comprehension, strong sum Σ and inductive generation. Elementary comprehension defines the set $\{x|E\}$ such that $x \in \{x|E\} \leftrightarrow E$ holds, provided the formula E is elementary. Sets introduced by elementary comprehension are all predicative because of elementary formulas. Elementary comprehension gives a wide class of sets which the usual set theory ZFC does not cover. Elementary comprehension is useful also in the area of computer science, since the sets introduced by it can represent expressively properties of programs. For example, the function type $\mathbf{N} \to \mathbf{N}$ of functions from natural numbers to natural numbers can be represented naturally by $\{f|\forall x(x \in \mathbf{N} \to fx \in \mathbf{N})\}$, which is impossible in ZFC since ZFC has only separation. Type theories can be interpreted by using constructive sets in variants of $\mathbf{T}_0$ [4].

(3) $\mathbf{T}_0$ is based on constructive logic so that it can be used for constructive programming. Constructive programming, which is also called program extraction, is a method of programming by which a program f and the proof of its correctness $\forall x(A(x) \to B(x, fx))$ are automatically produced from a constructive proof of its specification $\forall x(A(x) \to \exists y B(x, y))$, where x is an input, y is an output, $A(x)$ is an input condition and $B(x, y)$ is an input-output relation. By this method, we write a constructive proof to get a program instead of writing the program directly. The method integrates two separated steps: writing a program and verifying it, into the single step of writing a constructive proof. The method proposes a new programming paradigm based on the notion of proofs-as-programs. For example, the logical system **PX**, which is a variant of the theory $\mathbf{T}_0$, is used to formalize properties of LISP programs and to extract LISP programs from constructive proofs [6].

In the area of mathematical logic, the notion of realizability is an important tool to analyze constructive logics. Moreover **q**-realizability, which is a variant of realizability, is also important in the area of computer science, since the notion of **q**-realizability is the main

tool for constructive programming [6, 11, 13, 14, 8]. By using **q**-realizability, a program and its verification proof are automatically produced from a constructive proof of the specification of the program in the following way: from the constructive proof of $\forall x(A(x) \to \exists y B(x,y))$ and the soundness theorem of **q**-realizability, a realizer e is produced such that $(e \quad \mathbf{q} \quad \forall x(A(x) \to \exists y B(x,y)))$. According to the definition of **q**-realizability, the realizer e produces a program f and a proof of its correctness $\forall x(A(x) \to B(x, fx))$.

Until now, the only notion of **q**-realizability for $\mathbf{T}_0$ was the **q**-realizability interpretation given by Feferman and Beeson [3, 1]. But their interpretation doubles variables. In the area of mathematical logic, defining a **q**-realizability interpretation for $\mathbf{T}_0$ without doubling variables was left as an open problem.

The theory $\mathbf{T}_0$ is based on a two-sorted logic and has two kinds of variables: individual variables and set variables. Their interpretation doubles both kinds of variables. In [3], Feferman tried to give an interpretation which doubles only set variables and avoids doubling individual variables, but in a comment in [3], he states that he failed in giving such an interpretation, and that it is an open problem to find a **q**-realizability interpretation for the theory $\mathbf{T}_0$ that does not double individual variables.

In the area of computer science, it is also important to solve this problem since that interpretation cannot be used for constructive programming. Doubled variables bring garbage into programs extracted from constructive proofs via their interpretation. Therefore, in the area of computer science, it is also an important problem to give a notion of **q**-realizability which can avoid doubling variables and be used for constructive programming.

One has to use set completion to give a **q**-realizability interpretation for the theory. The completed set a^* of a set a is the set of terms (r,b) such that the formula $b \in a$ holds and the term r realizes the formula $b \in a$. For the soundness of the interpretation, elementary comprehension principle $x \in \{x|E\} \leftrightarrow E$ should be realized and a realizer of E should be produced from a realizer of $x \in \{x|E\}$. Therefore the interpretation of $a \in b$ has to use the completed set b^* and should be something like the formula $(r,a) \in b^*$.

Feferman and Beeson [3, 1] used the technique of doubling variables to get completed sets. The idea of doubling variables is that all the arguments given to a function are doubled so that the function gets also the completed set of each original argument as supplementary arguments. That is, an application fa of a function f to an argument a is transformed to the application faa^* in their interpretation. According to this idea, their interpretation doubled all the variables such as free variables and variables bound by λ, $\forall$ and $\exists$, and doubled also terms corresponding to variables. For example, we need to define $(\lambda x.t)^* \equiv \lambda x x^*.t^*$ and $(e \quad \mathbf{q} \quad \forall x A) \equiv \forall x x^*(e x x^* \quad \mathbf{q} \quad A)$.

The technique of doubling variables brings garbage into programs extracted from constructive proofs by their interpretation. When a formula is interpreted, every completed set for each term occurring in the formula is calculated in a realizer of the formula. Hence a program extracted from a constructive proof includes such calculation of completed sets. But almost all the calculation of the completed sets in the program is unnecessary to get the resulting value of the program. Such unnecessary calculation of the completed sets brings garbage into the program. For example, the program extracted by their interpretation from a constructive proof of the specification $\forall x \exists y(x = y)$ of the identify function is f such that $ftt^* = t$, instead of the usual indentity function I such that $It = t$. The program f needs an extra argument t^* because of variable doubling, though it is unnecessary for computing the result value t.

This paper solved this open problem. In this paper, we will give a **q**-realizability interpretation without any variable doubling for the theory $\mathbf{T}_0$ and prove its soundness. Moreover we prove a program extraction theorem, which guarantees that our interpretation enables us to use $\mathbf{T}_0$ for program synthesis. We will also explain examples of constructive programming by our interpretation. In the example, we will show that the usual indentity function I such that $It = t$ is extracted by our interpretation from its constructive proof, while the program extracted by the old interpretation from the same proof is f such that $ftt^* = t$.

The theory $\mathbf{T}_0$ we treat here is based on a one-sorted logic. If we use a two-sorted logic, our interpretation can avoid doubling not only individual variables but also set variables.

The basic idea used in our interpretation is to define a set completion program which computes the completed set a^* from a given set a by manipulating codes of sets. In the theory, all sets are intensional and given by their codes. By coding sets by terms which can be manipulated by programs, we enable set computation. Then it is possible to have a set completion program which computes the completed set from a given input. In this paper, we show that it is actually possible inside the theory. By our interpretation, a formula $a \in b$ is interpreted as a formula $(r,a) \in \mathrm{C}b$ where

the term C is the set completion program. Hence our interpretation avoids variable doubling.

Our interpretation enables us to synthesize programs using $\mathbf{T}_0$. By our interpretation, a realizer calculates the completed set a^* from a set a only when the completion a^* is necessary. Hence our interpretation does not bring unnecessary calculation of completed sets into the realizer, like Feferman and Beeson's approach did.

The system **PX**, which is a variant of $\mathbf{T}_0$, uses **q**-realizability to extract LISP programs from constructive proofs [6]. **PX** avoids doubling variables by restricting elementary comprehension to Harrop elementary formulas, that is, only a class of Harrop elementary formulas can introduce sets by elementary comprehension. Since Harrop formulas do not have any computational contents, the realizers of Harrop formulas are useless and can be defined to any value, for example, 0. In the same say, since sets introduced by Harrop elementary formulas do not have any computational contents, the realizers about these sets are useless and can be defined to any value. In particular, there is no need for set completion. In this way, **PX** gave up solving the problem and escaped from doubling variables. But, because of the additional restriction placed on elementary comprehension, the expressive power of comprehension is weaker in **PX** than in the original $\mathbf{T}_0$. For example, the following natural representations of cartesian product, the union of sets and the image of a function are prohibited in **PX** while they are possible in $\mathbf{T}_0$:

$$A \times B \equiv \{x | \exists yz(x = (y,z) \,\&\, y \in A \,\&\, z \in B\},$$
$$A \cup B \equiv \{x | x \in A \vee x \in B\},$$
$$f(A) \equiv \{y | \exists x \in A.y = fx\}.$$

Our **q**-realizability interpretation covers the interpretation given by Feferman and Beeson [1, 3], that is, our interpretation works well whenever their interpretation works. For example, independency results can be proved by our interpretation as well as by their interpretation.

The author gave a first interpretation in [12]. In this paper, we improve the results of that first version on the following points: (1) The previous version needed set induction to prove the soundness of realizability. Here we can prove the soundness without such additional induction. (2) We simplified the definition of **q**-realizability and the soundness proof by using the finite axiomatization of elementary comprehension instead of elementary comprehension axioms. In particular, the mutual recursive definition of **q**-realizability and set completion program used in [12] could be replaced by the usual definition of **q**-realizability.

The usual **r**-realizability cannot work for constructive programming, because provability of (e **r** A) does not imply the truth of A. For example, the axiom of choice $\mathbf{AC}_{\mathbf{Neg}}$ for negative formulas is **r**-realizable, but is not true in a classical sense. On the other hand, provability of (e **q** A) always implies the truth of A. Hence by **r**-realizability we cannot show that the correctness of programs is true in a classical sense.

The topic of avoiding garbage in program extraction has been studied intensively and widely, for example, in [6, 7, 9, 10]. A realizer has two kinds of contents, one is values contained in constructive proofs and the other is proofs which guarantee that the values are valid. Hence extracted programs include proofs which are unnecessary to compute result values. In general, unnecessary proofs in realizers bring garbage into extracted programs. Techniques using double negation and Harrop formulas [6], two propositional types *Spec* and *Prop* [10], subset types [9] and singleton, union and intersection types [7] were proposed to avoid unnecessary proofs in extracted programs. In this paper, we study how to avoid garbage brought by doubling variables and unnecessary calculation of completed sets for the specific case of **q**-realizability for the theory $\mathbf{T}_0$.

Section 2 presents our version of Feferman's theory $\mathbf{T}_0$. Section 3 defines the representation of elementary comprehensions. Section 4 presents the programming language used in the theory. Section 5 defines the **q**-realizability interpretation for the theory and proves the soundness of the interpretation. Section 6 proves the program extraction theorem and shows its application to program synthesis. This section also explains examples of constructive programming by our interpretation.

2 Feferman's Theory $\mathbf{T}_0$

Our version of the theory $\mathbf{T}_0$ is based on Beeson's [2]. That version differs from Feferman's original version [3] in that it is based on a one-sorted logic while the original version is based on a two-sorted logic.

Our version uses a finite axiomatization of elementary comprehension suggested by Feferman and based on [5]. In our version, the elementary comprehension principle is not included as axioms, but is derivable. The class of sets treated in our version is the same as that in the original version.

We will give our version of the theory $\mathbf{T}_0$ by modifying slightly the version of the theory $\mathbf{T}_0$ given in [2] on the points summarized as follows.

1. Decidable equality and a combinator **D** deciding

the equality are added. The combinator **d** is removed.

2. Axioms for definedness of normal term schemes are added.
3. Elementary comprehension axioms are replaced by a finite axiomatization of elementary comprehension. $\{x|E\}$ is not primitive and it is represented by other primitives **Eq**, **Inv**, **And**, **Or**, **Imp**, **All** and **Ex**.
4. The dependent product Π is added. This is essentially redundant since the dependent product Π can be defined by elementary comprehension and strong sum Σ.
5. Explicit coding of sets by terms is added.

The decidable equality, **D** and normal term schemes enable if-then-else statements which are used to write the set completion program. The finite axiomatization of elementary comprehension simplifies the definition of **q**-realizability. The coding of sets is needed for the set completion program.

Language

The language of the theory $\mathbf{T}_0$ is based on a usual first order language.

The constants are:

$$\mathbf{S},\quad \mathbf{K},\quad \mathbf{p},\quad \mathbf{p}_0,\quad \mathbf{p}_1,\quad 0,\quad \mathbf{s}_{\mathrm{N}},\quad \mathbf{p}_{\mathrm{N}},\quad \mathbf{D}.$$

We have only one function symbol **App** of arity 2.

A term t is defined by $t ::= x|c|\mathbf{App}(t,t)$ where x stands for a variable and c stands for a constant.

The target programming language of the theory is given by combinators with pairing and natural numbers. We have natural numbers as primitives, which are given by 0, the successor function $\mathbf{s}_{\mathrm{N}}$ and the predecessor function $\mathbf{p}_{\mathrm{N}}$. We also have the pairing functions $\mathbf{p}$, $\mathbf{p}_0$ and $\mathbf{p}_1$ as built-in, which correspond to `cons`, `car` and `cdr` in LISP respectively. **D** is a combinator judging equality of terms and corresponds to an if-then-else statement in a usual programming language.

The set $\mathrm{FV}(t)$ of all the free variables in the term t is defined as usual.

We use the following abbreviations: $ab \equiv \mathbf{App}(a,b)$, $(a,b) \equiv \mathbf{p}ab$, $n+1 \equiv \mathbf{s}_{\mathrm{N}}n$ for a natural number n, $(a_1,\ldots,a_n,a_{n+1}) \equiv (a_1,(\ldots,(a_n,a_{n+1})\ldots))$, $\mathbf{p}_{ij}t \equiv \mathbf{p}_j(\mathbf{p}_i t)$ for $i,j = 0,1$, $\mathbf{p}_{ijk}t \equiv \mathbf{p}_k(\mathbf{p}_j(\mathbf{p}_i t))$ for $i,j,k = 0,1$, $\mathbf{p}_i^n t \equiv \mathbf{p}_0(\underbrace{\mathbf{p}_1(\mathbf{p}_1\ldots(\mathbf{p}_1}_{i-1}t)\ldots))$ $(1 \le i < n)$, $\mathbf{p}_n^n t \equiv \underbrace{\mathbf{p}_1(\mathbf{p}_1\ldots(\mathbf{p}_1}_{n-1}t)\ldots)$.

For terms a,b and a variable x, $a_x[b]$ denotes the term obtained from a by replacing all the free occurrences of the variable x by b.

Atomic formulas are:

$$\bot,\quad a{\downarrow},\quad a=b,\quad \mathrm{S}(a),\quad a\in b.$$

$\bot$ means the contradiction. $a{\downarrow}$ means that the term a has a value. $a=b$ means that a equals b. $\mathrm{S}(a)$ means that the term a is the code of a set. $a\in b$ means that the term a is in the set coded by the term b.

A formula is constructed from atomic formulas by logical connectives $\&$, $\vee$, $\to$, $\forall$ and $\exists$ in the same way as the first order language. The set $\mathrm{FV}(A)$ of the free variables of the formula A is defined as usual.

Definition 2.1 (Harrop Formula). A formula A is called a *Harrop formula* iff neither $\mathrm{S}(a)$, $a\in b$,$\vee$ nor $\exists$ occurs strictly positively in A.

Definition 2.2 (Coding of Sets by Terms). Sets are coded by the following terms. Sets are represented by their codes. In the theory, formally the set given in the left hand side is an abbreviation of the term given in the right hand side.

$$\Sigma(a,b) \equiv (1,a,b),\quad \Pi(a,b) \equiv (2,a,b),$$
$$\mathbf{I}(a,b) \equiv (3,a,b),\quad \mathbf{N} \equiv 4,$$
$$\mathbf{Eq} \equiv 5,\quad \mathbf{Inv}(a,b) \equiv (6,a,b),$$
$$\mathbf{And}(a,b) \equiv (7,a,b),\quad \mathbf{Or}(a,b) \equiv (8,a,b),$$
$$\mathbf{Imp}(a,b) \equiv (9,a,b),\quad \mathbf{All}(a) \equiv (10,a),$$
$$\mathbf{Ex}(a) \equiv (11,a).$$

$\Sigma(a,b)$ means the set introduced by strong sum. $\Pi(a,b)$ means the set introduced by dependent product. $\mathbf{I}(a,b)$ means the set introduced by inductive generation. **N** means the set of natural numbers. **Eq**,...,**Ex** are used to give sets by elementary comprehension.

Definition 2.3 (Closed Normal Term). A *closed normal term* t is defined as follows:

$$t ::= \mathbf{p}t_1\ldots t_n|\mathbf{p}_0t_1\ldots t_n|\mathbf{p}_1t_1\ldots t_n$$
$$|0t_1\ldots t_n|\mathbf{s}_{\mathrm{N}}t_1\ldots t_n|\mathbf{p}_{\mathrm{N}}t_1\ldots t_n$$
$$|\mathbf{K}t_1|\mathbf{S}t_1t_2|\mathbf{D}t_1t_2t_3$$

where $t_1,\ldots,t_n$ and t range over closed normal terms and $n\ge 0$. In the terms $\mathbf{p}_0t_1\ldots t_n$ and $\mathbf{p}_1t_1\ldots t_n$, t_1 is not of the form $\mathbf{p}ab$. In the term $\mathbf{p}_{\mathrm{N}}t_1\ldots t_n$, t_1 is not a natural number m such that $m\ge 1$.

The universe of our intended model of the theory is the set of closed normal terms.

Definition 2.4 (Normal Term Scheme). A *normal term scheme* s is defined as follows:

$$s ::= x|\mathbf{p}x_1\ldots x_n|\mathbf{p}_0|\mathbf{p}_0sx_1\ldots x_n|\mathbf{p}_1|\mathbf{p}_1sx_1\ldots x_n$$
$$|0x_1\ldots x_n|\mathbf{s}_\mathrm{N}x_1\ldots x_n|\mathbf{p}_\mathrm{N}|\mathbf{p}_\mathrm{N}sx_1\ldots x_n$$
$$|\mathbf{K}x_1|\mathbf{S}x_1x_2|\mathbf{D}x_1x_2x_3$$

where $x_1,\ldots,x_n$ and x range over variables and s ranges over normal term schemes and $n \geq 0$. In the terms $\mathbf{p}_0sx_1\ldots x_n$ and $\mathbf{p}_1sx_1\ldots x_n$, s is not of the form $\mathbf{p}ab$. In the term $\mathbf{p}_\mathrm{N}sx_1\ldots x_n$, s is not a natural number m such that $m \geq 1$.

The set of closed normal terms is characterized by the least set X such that X includes all terms obtained from normal term schemes by substituting terms in X for all the free variables.

Axioms and Inference Rules

x,y,z,w range over variables and a,b,c range over terms.

Axioms for partial terms:

(V1) $t\!\downarrow$ (t is a normal term scheme), (V2) $ab\!\downarrow \to a\!\downarrow \,\&\, b\!\downarrow$, (V3) $a = b \to a\!\downarrow \,\&\, b\!\downarrow$, (V4) $\mathrm{S}(a) \to a\!\downarrow$, (V5) $a \in b \to a\!\downarrow \,\&\, b\!\downarrow$.

$(a\!\downarrow \to a = b) \,\&\, (b\!\downarrow \to a = b)$ is abbreviated as $a \simeq b$.

The logical axioms and inference rules are the same as those of usual intuitionistic logic except for the following two quantifier rules.

$$\frac{\forall xA(x) \quad t\!\downarrow}{A(t)}\ (\forall E) \qquad \frac{A(t) \quad t\!\downarrow}{\exists xA(x)}\ (\exists I)$$

Axioms for equality: (E1) $x = x$, (E2) $x \simeq y \,\&\, A(x) \to A(y)$.

Axioms for combinators: (C1) $\mathbf{K}xy = x$, (C2) $\mathbf{S}xyz \simeq xz(yz)$.

Axioms for pairing: (P1) $\mathbf{p}_0(\mathbf{p}xy) = x$, (P2) $\mathbf{p}_1(\mathbf{p}xy) = y$.

Axioms for natural numbers: (N1) $0 \in \mathbf{N}$, (N2) $x \in \mathbf{N} \to \mathbf{s}_\mathrm{N}x \in \mathbf{N}$, (N3) $x \in \mathbf{N} \to \mathbf{p}_\mathrm{N}(\mathbf{s}_\mathrm{N}x) = x$, (N4) $x \in \mathbf{N} \to \mathbf{s}_\mathrm{N}x \neq 0$, (N5) $A(0) \,\&\, \forall x(x \in \mathbf{N} \,\&\, A(x) \to A(\mathbf{s}_\mathrm{N}x)) \to \forall x(x \in \mathbf{N} \to A(x))$.

Axioms and inference rules for decidable equality: (D1) $x = y \vee x \neq y$, (D2) $x = y \to \mathbf{D}xyzw = z$, (D3) $x \neq y \to \mathbf{D}xyzw = w$, (D4) $cs_1\ldots s_n \neq dt_1\ldots t_m$ where $cs_1\ldots s_n$, $dt_1\ldots t_m$,$s_1,\ldots,s_n$,$t_1,\ldots,t_m$ are normal term schemes, c and d are constants, $n,m \geq 0$ and (1) $c \not\equiv d$ or (2) $n \neq m$ or (3) $n = m$,$1 \leq i \leq n$ and $s_i \neq t_i$ for some i.

Axioms for sets:

$$x \in y \to \mathrm{S}(y), \tag{S1}$$
$$\mathrm{S}(\mathbf{Eq}), \tag{Eq1}$$
$$\forall x(x \in \mathbf{Eq} \leftrightarrow \mathbf{p}_0x = \mathbf{p}_1x), \tag{Eq2}$$
$$\mathrm{S}(y) \leftrightarrow \mathrm{S}(\mathbf{Inv}(x,y)), \tag{Inv1}$$
$$\mathrm{S}(\mathbf{Inv}(x,y)) \to \forall z.(z \in \mathbf{Inv}(x,y) \leftrightarrow xz \in y), \tag{Inv2}$$
$$\mathrm{S}(x) \,\&\, \mathrm{S}(y) \leftrightarrow \mathrm{S}(\mathbf{And}(x,y)), \tag{And1}$$
$$\mathrm{S}(\mathbf{And}(x,y)) \to \forall z.(z \in \mathbf{And}(x,y) \leftrightarrow z \in x \,\&\, z \in y), \tag{And2}$$
$$\mathrm{S}(x) \,\&\, \mathrm{S}(y) \leftrightarrow \mathrm{S}(\mathbf{Or}(x,y)), \tag{Or1}$$
$$\mathrm{S}(\mathbf{Or}(x,y) \to \forall z.(z \in \mathbf{Or}(x,y) \leftrightarrow z \in x \vee z \in y), \tag{Or2}$$
$$\mathrm{S}(x) \,\&\, \mathrm{S}(y) \leftrightarrow \mathrm{S}(\mathbf{Imp}(x,y)), \tag{Imp1}$$
$$\mathrm{S}(\mathbf{Imp}(x,y) \to \forall z.(z \in \mathbf{Imp}(x,y) \leftrightarrow (z \in x \to z \in y)), \tag{Imp2}$$
$$\mathrm{S}(x) \leftrightarrow \mathrm{S}(\mathbf{All}(x)), \tag{All1}$$
$$\mathrm{S}(\mathbf{All}(x)) \to \forall y.(y \in \mathbf{All}(x) \leftrightarrow \forall z.((z,y) \in x)), \tag{All2}$$
$$\mathrm{S}(x) \leftrightarrow \mathrm{S}(\mathbf{Ex}(x)), \tag{Ex1}$$
$$\mathrm{S}(\mathbf{Ex}(x)) \to \forall y.(y \in \mathbf{Ex}(x) \leftrightarrow \exists z.((z,y) \in x)), \tag{Ex2}$$
$$\mathrm{S}(x) \,\&\, \forall z \in x.S(yz) \leftrightarrow \mathrm{S}(\Sigma(x,y)), \tag{Σ1}$$
$$\mathrm{S}(\Sigma(x,y)) \to \forall z(z \in \Sigma(x,y) \leftrightarrow \mathbf{p}_0z \in x \,\&\, \mathbf{p}_1z \in y(\mathbf{p}_0z)), \tag{Σ2}$$
$$\mathrm{S}(x) \,\&\, \forall z \in x.S(yz) \leftrightarrow \mathrm{S}(\Pi(x,y)), \tag{Π1}$$
$$\mathrm{S}(\Pi(x,y)) \to \forall f(f \in \Pi(x,y) \leftrightarrow \forall z \in x.fz \in yz), \tag{Π2}$$
$$\mathrm{S}(x) \,\&\, \mathrm{S}(y) \leftrightarrow \mathrm{S}(\mathbf{I}(x,y)), \tag{IG1}$$
$$\mathrm{S}(\mathbf{I}(x,y)) \to \forall z.(\forall w.((w,z) \in y \to w \in \mathbf{I}(x,y)) \to z \in \mathbf{I}(x,y)), \tag{IG2}$$
$$\mathrm{S}(\mathbf{I}(x,y)) \to \forall z \in x.(\forall w.((w,z) \in y \to A(w)) \to A(z)) \to \forall z \in \mathbf{I}(x,y).A(z). \tag{IG3}$$

The axioms (**Eq**1), ..., (**Ex**2) give a finite axiomatization of elementary comprehension. The axioms (Σ1) and (Σ2) define sets introduced by strong sum. The axioms (Π1) and (Π2) define sets introduced by dependent product. The axioms (IG1),(IG2) and (IG3) define sets introduced by inductive generation.

3 Representation of Elementary Comprehension

We will give a representation of elementary comprehension and prove that elementary comprehension axioms in the original version of $\mathbf{T}_0$ are derivable in the version of $\mathbf{T}_0$ we defined in the previous section.

Definition 3.1 (Elementary Formula). A formula E is called an *elementary formula* iff

1. E does not have any subformula of the form $\mathrm{S}(a)$,

2. if E has a subformula $a \in b$, then all the free occurrence of variables in b are also free in E.

A term b such that an elementary formula E has the subformula $a \in b$ is called an *$\in$-right term* of E. $\mathrm{ER}(E)$ denotes the set of $\in$-right terms of the elementary formula E.

The definition of elementary formulas is slightly different from the original definition [1, 3] but the class of sets introduced by elementary comprehension in the theory given in this paper is the same.

$\mathrm{FV}(\mathrm{ER}(E))$ denotes $\mathrm{FV}(b_1) \cup \ldots \cup \mathrm{FV}(b_k)$ and $\mathrm{S}(\mathrm{ER}(E))$ denotes $\mathrm{S}(b_1)\ \&\ \ldots\ \&\ \mathrm{S}(b_k)$ when $\mathrm{ER}(E) = \{b_1, \ldots, b_k\}$.

In this section, we will show that the elementary comprehension principle is derivable, that is, $\mathrm{S}(\mathrm{ER}(E)) \leftrightarrow \mathrm{S}(\{x|E\})$ and $\mathrm{S}(\{x|E\}) \rightarrow \forall x(x \in \{x|E\} \leftrightarrow E)$ hold in our version of $\mathbf{T}_0$.

Definition 3.2 ($\mathrm{EC}_E^{x_1 \ldots x_n}$). For an elementary formula E and variables $x_1 \ldots x_n \notin \mathrm{ER}(E)$, a term $\mathrm{EC}_E^{x_1 \ldots x_n}$ is defined by induction on E:

$\mathrm{EC}_{a=b}^{x_1 \ldots x_n} \equiv$
$\quad \mathbf{Inv}(\lambda x.(\lambda x_1 \ldots x_n.(a,b))(\mathbf{p}_1^n x) \ldots (\mathbf{p}_n^n x), \mathbf{Eq}),$
$\mathrm{EC}_{a \in b}^{x_1 \ldots x_n} \equiv$
$\quad \mathbf{Inv}(\lambda x.(\lambda x_1 \ldots x_n.a)(\mathbf{p}_1^n x) \ldots (\mathbf{p}_n^n x), b),$
$\mathrm{EC}_{a\downarrow}^{x_1 \ldots x_n} \equiv \mathrm{EC}_{a=a}^{x_1 \ldots x_n},$
$\mathrm{EC}_{\bot}^{x_1 \ldots x_n} \equiv \mathbf{Inv}(\lambda x.(0,1), \mathbf{Eq}),$
$\mathrm{EC}_{A\&B}^{x_1 \ldots x_n} \equiv \mathbf{And}(\mathrm{EC}_A^{x_1 \ldots x_n}, \mathrm{EC}_B^{x_1 \ldots x_n}),$
$\mathrm{EC}_{A \vee B}^{x_1 \ldots x_n} \equiv \mathbf{Or}(\mathrm{EC}_A^{x_1 \ldots x_n}, \mathrm{EC}_B^{x_1 \ldots x_n}),$
$\mathrm{EC}_{A \to B}^{x_1 \ldots x_n} \equiv \mathbf{Imp}(\mathrm{EC}_A^{x_1 \ldots x_n}, \mathrm{EC}_B^{x_1 \ldots x_n}),$
$\mathrm{EC}_{\forall x A x}^{x_1 \ldots x_n} \equiv \mathbf{All}(\mathrm{EC}_A^{x x_1 \ldots x_n}),$
$\mathrm{EC}_{\exists x A x}^{x_1 \ldots x_n} \equiv \mathbf{Ex}(\mathrm{EC}_A^{x x_1 \ldots x_n}).$

We use the following abbreviations:
$\{(x_1, \ldots, x_n)|E\} \equiv \mathrm{EC}_E^{x_1 \ldots x_n} \quad (n \geq 1),$
$\{x|E\} \equiv \mathrm{EC}_E^{x}.$

Note that in the sequel, E is an elementary formula and $x \notin \mathrm{FV}(\mathrm{ER}(E))$ holds whenever we use the notation $\{x|E\}$.

Theorem 3.3. *For an elementary formula E, $n \geq 1$, $x_1, \ldots, x_n \notin FV(ER(E))$, the followings hold.*
(1) $\mathrm{S}(ER(E)) \leftrightarrow \mathrm{S}(\{(x_1, \ldots, x_n)|E\})$.
(2) $\mathrm{S}(\{(x_1, \ldots, x_n)|E\}) \rightarrow \forall x_1 \ldots x_n($
$\quad (x_1, \ldots, x_n) \in \{(x_1, \ldots, x_n)|E\} \leftrightarrow E)$.

Corollary 3.4 (Elementary Comprehension). *The followings hold in the theory.*
$\mathrm{S}(ER(E)) \leftrightarrow \mathrm{S}(\{x|E\}),$ *(ECA1)*
$\mathrm{S}(\{x|E\}) \rightarrow \forall x(x \in \{x|E\} \leftrightarrow E).$ *(ECA2)*

This shows that that elementary comprehension axioms (ECA1) and (ECA2) in the original version of $\mathbf{T}_0$ are derivable in our version of $\mathbf{T}_0$.

4 A Programming Language Based on $\mathbf{T}_0$

We will write programs by using terms of $\mathbf{T}_0$. For this purpose, we introduce some syntactic sugar for terms of $\mathbf{T}_0$, to write programs in a fashion similar to usual functional programming languages.

Definition 4.1 (λ-Term). A term $\overline{t}$ of $\mathbf{T}_0$ is defined from a term t which contains λ constructors in the following way:

1. $\overline{x} \equiv x$ if x is a variable,
2. $\overline{c} \equiv c$ if c is a constant,
3. $\overline{t_1\, t_2} \equiv \overline{t}_1\, \overline{t}_2$,
4. $\overline{\lambda x.x} \equiv \mathbf{SKK}$,
5. $\overline{\lambda x.y} \equiv \mathbf{K}y$ if y is a variable and $x \not\equiv y$,
6. $\overline{\lambda x.c} \equiv \mathbf{K}c$ if c is a constant,
7. $\overline{\lambda x.t_1 t_2} \equiv \mathbf{S}(\overline{\lambda x.t_1})(\overline{\lambda x.t_2})$.

Notation 4.2. In the sequel, a term t always denotes a term $\overline{t}$ of $\mathbf{T}_0$ except that a substituted term $t_1[t_2/x]$ denotes a term $\overline{t}_1[\overline{t}_2/x]$ of $\mathbf{T}_0$ for terms t_1,t_2 and a variable x. Note that $t_1[t_2/x]$ does not necessarily denote $\overline{t_1[t_2/x]}$ since $\overline{t}_1[\overline{t}_2/x] \equiv \overline{t_1[t_2/x]}$ does not necessarily hold.

Then the following claims hold.

Proposition 4.3. *(1) $\lambda x.t\downarrow$,*
(2) $s\downarrow \rightarrow (\lambda x.t)s \simeq t[s/x]$.

Definition 4.4 (If-then-else Statement). For terms a,b,c and d,
$\quad \mathtt{if}\ a = b\ \mathtt{then}\ c\ \mathtt{else}\ d \equiv \mathbf{D}ab(\lambda r.c)(\lambda r.d)0$
where r is a fresh variable.

Then the following claims hold.

Proposition 4.5. *For variables x and y and terms a and b,*
(1) $x = y \rightarrow (\mathtt{if}\ x = y\ \mathtt{then}\ a\ \mathtt{else}\ b) \simeq a$,
(2) $x \neq y \rightarrow (\mathtt{if}\ x = y\ \mathtt{then}\ a\ \mathtt{else}\ b) \simeq b$.

We want to use also the following let statement:
$\quad \mathtt{let}\ p_1 = v_1, \ldots, p_n = v_n\ \mathtt{in}\ a\ \mathtt{else}\ b$
where $p_1, \ldots, p_n$ are patterns.

Definition 4.6. A pattern p is defined as follows:
$\quad p ::= x|n|(p_1, p_2)$
where x stands for a variable, n stands for a natural number and p,p_1 and p_2 stand for patterns. In the clause (p_1,p_2), we suppose that $\mathrm{FV}(p_1) \cap \mathrm{FV}(p_2) = \phi$.

We can define the representation of the let statement $\texttt{let}\ p_1 = v_1, \ldots, p_n = v_n\ \texttt{in}\ a\ \texttt{else}\ b$ in $\mathbf{T}_0$ straightforwardly by induction on n. We omit the details of the definition here. Then the next claim holds.

Proposition 4.7. *Let $FV(p_i) = x_i$ where x_i denotes variables.*

$$\exists x_1 \ldots x_n (p_1 = y_1 \,\&\, \ldots \,\&\, p_n = y_n \,\&\, (\texttt{let}\ p_1 = y_1, \ldots, p_n = y_n\ \texttt{in}\ a\ \texttt{else}\ b) \simeq a) \vee (\neg\exists x_1 (p_1 = y_1) \vee \ldots \vee \neg\exists x_n (p_n = y_n)) \,\&\, (\texttt{let}\ p_1 = y_1, \ldots, p_n = y_n\ \texttt{in}\ a\ \texttt{else}\ b) \simeq b).$$

Theorem 4.8 (Recursion Theorem). *There exists a closed normal term R such that*
(1) $R(\lambda f x_1 \ldots x_n.t) x_1 \ldots x_{n-1}\downarrow$,
(2) $R(\lambda f x_1 \ldots x_n.t) x_1 \ldots x_n \simeq (\lambda f x_1 \ldots x_n.t)(R(\lambda f x_1 \ldots x_n.t)) x_1 \ldots x_n$.

Proof 4.9. Let $V \equiv \lambda y x.f(yy)x$ and $R \equiv \lambda f.VV$. □

$\mu f.t$ denotes $R(\lambda f.t)$.

5 q-Realizability for $\mathbf{T}_0$

We will define a **q**-realizability interpretation for $\mathbf{T}_0$. In the definition, we use a program C which computes the code of the completed set of a set A from the code of the set A.

Definition 5.1 (q-Realizability). We define the set completion program C given in Figure 1.

Note that in the above definition we use abbreviations of codes of sets and sets introduced by elementary comprehension. For example, the line of $\mathbf{Inv}(x,y)$ denotes the following:

$$\texttt{else let}\ (6,x,y) = s\ \texttt{in}\ (7, (6, \lambda x.(\lambda e z.(e, xz))(\mathbf{p}_0 x)(\mathbf{p}_1 x), cy), (6, \lambda x.(\lambda e z.z)(\mathbf{p}_0 x)(\mathbf{p}_1 x), (6, x, y)))$$

We define $(e\ \ \mathbf{q}\ \ A)$ as follows:

$(e\ \ \mathbf{q}\ \ A) \equiv e = 0 \,\&\, A$ if A is a Harrop formula,
$(e\ \ \mathbf{q}\ \ \mathrm{S}(a)) \equiv e = 0 \,\&\, \mathrm{S}(a) \,\&\, \mathrm{S}(\mathrm{C}a)$,
$(e\ \ \mathbf{q}\ \ a \in b) \equiv (e, a) \in \mathrm{C}b \,\&\, a \in b$,
$(e\ \ \mathbf{q}\ \ A \,\&\, B) \equiv (\mathbf{p}_0 e\ \ \mathbf{q}\ \ A) \,\&\, (\mathbf{p}_1 e\ \ \mathbf{q}\ \ B)$,
$(e\ \ \mathbf{q}\ \ A \vee B) \equiv \mathbf{p}_0 e \in \mathbf{N} \,\&\, (\mathbf{p}_0 e = 0 \to (\mathbf{p}_1 e\ \ \mathbf{q}\ \ A)) \,\&\, (\mathbf{p}_0 e \neq 0 \to (\mathbf{p}_1 e\ \ \mathbf{q}\ \ B))$,
$(e\ \ \mathbf{q}\ \ A \to B) \equiv \forall q((q\ \ \mathbf{q}\ \ A) \to (eq\ \ \mathbf{q}\ \ B)) \,\&\, (A \to B) \,\&\, e\downarrow$,
$(e\ \ \mathbf{q}\ \ \forall x A(x)) \equiv \forall x (ex\ \ \mathbf{q}\ \ A(x))$,
$(e\ \ \mathbf{q}\ \ \exists x A(x)) \equiv (\mathbf{p}_1 e\ \ \mathbf{q}\ \ A(\mathbf{p}_0 e))$.

Proposition 5.2. *The properties given in Figure 2 hold.*

Proposition 5.3. *(1) $(e\ \ \mathbf{q}\ \ A)_x[t] \equiv (e_x[t]\ \ \mathbf{q}\ \ A_x[t])$.*
(2) $(e\ \ \mathbf{q}\ \ A) \to e\downarrow$.
(3) $(e\ \ \mathbf{q}\ \ A) \to A$.

Proposition 5.4. *(1) If A is $(\Sigma 2)$, $(\Pi 2)$, $(\mathbf{Inv}2)$, $(\mathbf{And}2)$, $(\mathbf{Or}2)$, $(\mathbf{Imp}2)$, $(\mathbf{All}2)$ or $(\mathbf{Ex}2)$, then $\lambda qz.(\lambda r.r, \lambda r.r)\ \ \mathbf{q}\ \ A$.*
(2) $\lambda x.(\lambda r.r, \lambda r.r)\ \ \mathbf{q}\ \ (\mathbf{Eq}2)$.
(3) $\lambda qzr.r\ \ \mathbf{q}\ \ (IG2)$.
(4) If $A(x)$ is not a Harrop formula, $(\lambda pq.\mu f.\lambda zr.qzr(\lambda xy.fx(rxy))\ \ \mathbf{q}\ \ (IG3))$ holds.

Theorem 5.5 (Soundness of q-Realizability). *If $\mathbf{T}_0 \vdash A$ holds, then we can get a term e effectively from the proof and $\mathbf{T}_0 \vdash (e\ \ \mathbf{q}\ \ A)$ holds.*

The realizability gives the same realizers to an elementary formula E and the associated membership $x \in \{x|E\}$. The next corollary shows it.

Theorem 5.6.
$(e, x_1, \ldots, x_n) \in C\{(x_1, \ldots, x_n)|E\} \leftrightarrow (e\ \ \mathbf{q}\ \ E)$.

Corollary 5.7. *(1) $(e, x) \in C\{x|E\} \leftrightarrow (e\ \ \mathbf{q}\ \ E)$.*
(2) If E is not Harrop, $(\lambda qx.(\lambda r.r, \lambda r.r)\ \ \mathbf{q}\ \ (ECA2))$ holds.

6 Application to Program Synthesis

By using the soundness theorem, we will prove a program extraction theorem, which means that by our realizability interpretation $\mathbf{T}_0$ can be used for program extraction. We will show that our interpretation can be used for program extraction while the interpretation given by Feferman and Beeson cannot be used for program extraction. Examples of constructive programming in $\mathbf{T}_0$ will be given.

Theorem 6.1 (Program Extraction).
Suppose that $A(x)$ and $B(x,y)$ are formulas and $\mathbf{T}_0 \vdash \forall x(A(x) \to (jx\ \ \mathbf{q}\ \ A(x)))$ holds for some term j. If we prove a specification formula $\forall x(A(x) \to \exists y B(x,y))$ in the theory $\mathbf{T}_0$, then we can get a program f effectively from the proof and $\mathbf{T}_0 \vdash \forall x(A(x) \to fx\downarrow \,\&\, B(x, fx))$ holds.

Proof 6.2. By Theorem 5.5, we can get a term e such that $\mathbf{T}_0 \vdash (e\ \ \mathbf{q}\ \ \forall x(A(x) \to \exists y B(x,y)))$. Let $f \equiv \lambda x.\mathbf{p}_0(ex(jx))$. Then the claim holds. □

$$
\begin{array}{l}
\mathrm{C} \equiv \mu c.\lambda s. \\
\texttt{let } \mathbf{Eq} = s \texttt{ in } \{(e,x)|e = 0 \,\&\, x \in \mathbf{Eq}\} \\
\texttt{else let } \mathbf{Inv}(x,y) = s \texttt{ in } \{(e,z)|(e,xz) \in cy \,\&\, z \in \mathbf{Inv}(x,y)\} \\
\texttt{else let } \mathbf{And}(x,y) = s \texttt{ in } \{(e,z)|(e_0,z) \in cx \,\&\, (e_1,z) \in cy \,\&\, z \in \mathbf{And}(x,y)\} \\
\texttt{else let } \mathbf{Or}(x,y) = s \texttt{ in } \{(e,z)|e_0 \in \mathbf{N} \,\&\, (e_0 = 0 \to (e_1,z) \in cx \,\&\, z \in x) \,\& \\
\quad (e_0 \neq 0 \to (e_1,z) \in cy \,\&\, z \in y)\} \\
\texttt{else let } \mathbf{Imp}(x,y) = s \texttt{ in } \{(e,z)|\forall w((w,z) \in cx \,\&\, z \in x \to (ew,z) \in cy) \,\&\, z \in \mathbf{Imp}(x,y)\} \\
\texttt{else let } \mathbf{All}(x) = s \texttt{ in } \{(e,y)|\forall z((ez,z,y) \in cx) \,\&\, y \in \mathbf{All}(x)\} \\
\texttt{else let } \mathbf{Ex}(x) = s \texttt{ in } \{(e,y)|(e_1,e_0,y) \in cx \,\&\, (e_0,y) \in x\} \\
\texttt{else let } \mathbf{N} = s \texttt{ in } \{(e,x)|e = 0 \,\&\, x \in \mathbf{N}\} \\
\texttt{else let } \Sigma(x,y) = s \texttt{ in } \{(r,z)|((\mathbf{p}_0 r, \mathbf{p}_0 z),(\mathbf{p}_1 r, \mathbf{p}_1 z)) \in \\
\quad \Sigma(\{z|z \in cx \,\&\, \mathbf{p}_1 z \in x\}, \lambda x.c(y(\mathbf{p}_1 x))) \,\&\, z \in \Sigma(x,y)\} \\
\texttt{else let } \Pi(x,y) = s \texttt{ in } \{(r,f)|\lambda z.(r(\mathbf{p}_1 z)(\mathbf{p}_0 z), f(\mathbf{p}_1 z)) \in \\
\quad \Pi(\{z|z \in cx \,\&\, \mathbf{p}_1 z \in x\}, \lambda z.c(y(\mathbf{p}_1 z))) \,\&\, f \in \Pi(x,y) \,\&\, \forall z(rz\!\downarrow)\} \\
\texttt{else let } \mathbf{I}(x,y) = s \texttt{ in } \{(r,z)|(r,z) \in \mathbf{I}(\{(r,z)|(r,z) \in cx \,\&\, z \in x\}, \\
\quad \{((s,w),(r,z))|\exists q(s = rwq \,\&\, (q,w,z) \in cy) \,\&\, (w,z) \in y\}) \,\&\, z \in \mathbf{I}(x,y)\} \\
\texttt{else } 0.
\end{array}
$$

Figure 1: Set Completion Program

$$
\begin{array}{l}
C\mathbf{Eq} = \{(e,x)|e = 0 \,\&\, x \in \mathbf{Eq}\}, \\
C(\mathbf{Inv}(x,y)) = \{(e,z)|(e,xz) \in Cy \,\&\, z \in \mathbf{Inv}(x,y)\}, \\
C(\mathbf{And}(x,y)) = \{(e,z)|(e_0,z) \in Cx \,\&\, (e_1,z) \in Cy \,\&\, z \in \mathbf{And}(x,y)\}, \\
C(\mathbf{Or}(x,y)) = \{(e,z)|e_0 \in \mathbf{N} \,\&\, (e_0 = 0 \to (e_1,z) \in Cx \,\&\, z \in x) \,\&\, (e_0 \neq 0 \to (e_1,z) \in Cy \,\&\, z \in y)\}, \\
C(\mathbf{Imp}(x,y)) = \{(e,z)|\forall w((w,z) \in Cx \,\&\, z \in x \to (ew,z) \in Cy) \,\&\, z \in \mathbf{Imp}(x,y)\}, \\
C(\mathbf{All}(x)) = \{(e,y)|\forall z((ez,z,y) \in Cx) \,\&\, y \in \mathbf{All}(x)\}, \\
C(\mathbf{Ex}(x)) = \{(e,y)|(e_1,e_0,y) \in Cx \,\&\, (e_0,y) \in x\}, \\
C\mathbf{N} = \{(e,x)|e = 0 \,\&\, x \in \mathbf{N}\}, \\
C(\Sigma(x,y)) = \{(r,z)|((\mathbf{p}_0 r, \mathbf{p}_0 z),(\mathbf{p}_1 r, \mathbf{p}_1 z)) \in \\
\quad \Sigma(\{z|z \in Cx \,\&\, \mathbf{p}_1 z \in x\}, \lambda x.C(y(\mathbf{p}_1 x))) \,\&\, z \in \Sigma(x,y)\}, \\
C(\Pi(x,y)) = \{(r,f)|\lambda z.(r(\mathbf{p}_1 z)(\mathbf{p}_0 z), f(\mathbf{p}_1 z)) \in \\
\quad \Pi(\{z|z \in Cx \,\&\, \mathbf{p}_1 z \in x\}, \lambda z.C(y(\mathbf{p}_1 z))) \,\&\, f \in \Pi(x,y) \,\&\, \forall z(rz\!\downarrow)\}, \\
C(\mathbf{I}(x,y)) = \{(r,z)|(r,z) \in \mathbf{I}(\{(r,z)|(r,z) \in Cx \,\&\, z \in x\}, \\
\quad \{((s,w),(r,z))|\exists q(s = rwq \,\&\, (q,w,z) \in Cy) \,\&\, (w,z) \in y\}) \,\&\, z \in \mathbf{I}(x,y)\}.
\end{array}
$$

Figure 2: Properties

The condition $\forall x(A(x) \to (jx \;\; \mathbf{q} \;\; A(x)))$ means that the formula $A(x)$ is self-realizing. Harrop formulas satisfy this condition.

We will explain that our interpretation is necessary for program extraction in $\mathbf{T}_0$ and that the interpretation given by Feferman and Beeson [1, 3] cannot be used for this.

For the interpretation given by Feferman and Beeson, only a weak form of the program extraction theorem holds. We use the notation $(e \;\; \mathbf{q}^F \;\; A)$ to denote the interpretation given by Feferman and Beeson. The program extraction theorem proved by using their interpretation is as follows.

Suppose that $A(x)$ and $B(x,y)$ are formulas and $\mathbf{T}_0 \vdash \forall x(A(x) \to (jx \quad \mathbf{q}^F \quad A(x)))$ holds for some term j. If we prove a specification formula $\forall x(A(x) \to \exists y B(x,y))$ in the theory $\mathbf{T}_0$, then we can get a program f effectively from the proof and for any term t, $\mathbf{T}_0 \vdash A(t) \to ftt^*\downarrow \,\&\, B(t, ftt^*)$ holds where t^* denotes the completion of the term t.

The point is that the program extracted by their interpretation needs the extra argument t^* besides the argument t. To compute the value of the program, we need to calculate the term t^* before we execute the program f with the input t. This comes from the fact that the formula $(e \quad \mathbf{q}^F \quad \forall x(A(x) \to \exists y B(x,y)))$ is defined as

$$\forall x x^* q(A(x) \,\&\, (q \quad \mathbf{q}^F \quad A(x)) \to \exists y y^*(\mathbf{p}_1(exx^*q) = (y, y^*) \,\&\, B(x,y) \,\&\, (\mathbf{p}_0(exx^*q) \quad \mathbf{q}^F \quad B(x,y)))).$$

Example 6.3. We will give an example of synthesis for the identity function. Suppose that we want to produce the identity function on all elements. Then the specification formula of the program is given by the formula $\forall x \exists y(x = y)$. Suppose that we prove the specification formula in $\mathbf{T}_0$ in the usual way. By Theorem 5.5, we have the realizer $\lambda x.(x,0)$ and $(\lambda x.(x,0) \quad \mathbf{q} \quad \forall x \exists y(x = y))$ holds. By Theorem 6.1, we have the desired program f by $f \equiv \lambda x.\mathbf{p}_0((\lambda x.(x,0))x)$.

On the other hand, by the soundness theorem of the interpretation given by Feferman and Beeson, we have the realizer $\lambda x x^*.(0,(x,x^*))$ and $(\lambda x x^*.(0,(x,x^*)) \quad \mathbf{q}^F \quad \forall x \exists y(x = y))$ holds. Then the extracted program is $\lambda x x^*.\mathbf{p}_{10}((\lambda x x^*.(0,(x,x^*)))xx^*)$. The program needs the extra argument x^* besides the argument x, though the argument x^* is not used to compute the resulting value x of the program. It comes from the fact that by the definition of their interpretation, the formula $(e \quad \mathbf{q}^F \quad \forall x \exists y(x = y))$ is defined as $\forall x x^* \exists y y^*(\mathbf{p}_1(exx^*) = (y, y^*) \,\&\, x = y \,\&\, x^* = y^*)$.

Example 6.4. We will give an example of synthesis for the characteristic function of the union of given two sets. Suppose that subsets A and B of $\mathbf{N}$ and their characteristic functions F_A and F_B are given. Let $\mathrm{Char}(F,X)$ be $\forall x \in \mathbf{N}((Fx = 0 \vee Fx = 1) \& (Fx = 0 \to x \notin X) \,\&\, (Fx = 1 \to x \in X))$. Suppose that $\mathrm{Char}(F_A, A)$ and $\mathrm{Char}(F_B, B)$ are proved. Let

$$\mathrm{Union} \equiv \lambda xy.\{z | z \in x \vee z \in y\}$$

and $A \cup B$ be $\mathrm{Union}\, A\, B$. The specification formula of the characteristic function of $A \cup B$ is

$$\forall x \in \mathbf{N} \exists y(x \notin A \cup B \,\&\, y = 0 \vee x \in A \cup B \,\&\, y = 1).$$

Note that $A \cup B$ is not available in **PX** while it can be used in full $\mathbf{T}_0$.

We can prove the specification formula in the following way. First, cases are classified by $F_A x = 0$ or $F_A x = 1$. In the first case, $x \notin A$ holds and cases are classified again by $F_B x = 0$ or $F_B x = 1$. If $F_B x = 0$ holds, then $x \notin B$ holds and the claim is proved by putting $y = 0$. If $F_B x = 1$ holds, then $x \in B$ holds and the claim is proved by putting $y = 1$. If the second case, $x \in A$ holds and the claim is proved by putting $y = 1$.

Let e_A be a realizer of $\mathrm{Char}(F_A, A)$ and e_B be a realizer of $\mathrm{Char}(F_B, B)$. The realizer e corresponding to the proof is

$$\begin{aligned}&\lambda xr.\ \texttt{if}\ (e_A x0)_{00} = 0\ \texttt{then}\\ &\quad (\texttt{if}\ (e_B x0)_{00} = 0\ \texttt{then}\ (0,(0,0))\\ &\quad \texttt{else}\ (1,(1,((1,(e_B x0)_{11}0),0))))\\ &\quad \texttt{else}\ (1,(1,((0,(e_A x0)_{11}0),0))).\end{aligned}$$

By Theorem 6.1, the extracted program G is $\lambda x.(ex0)_0$. By transforming it by using the induction on x, $(e_A x0)_{00} = 0 \leftrightarrow F_A x = 0$, and $(e_B x0)_{00} = 0 \leftrightarrow F_B x = 0$, we can get the expected program

$$\begin{aligned}&Gx = \ \texttt{if}\ F_A x = 0\ \texttt{then}\\ &\quad (\texttt{if}\ F_B x = 0\ \texttt{then}\ 0\ \texttt{else}\ 1)\ \texttt{else}\ 1\end{aligned}$$

for $x \in \mathbf{N}$.

Acknowledgment

I would like to thank Professor Solomon Feferman for valuable discussions, helpful comments and encouragement. I would also like to thank Mr. Satoshi Kobayashi for discussions and Dr. Jacques Garrigue for advice on expressions.

References

[1] M. Beeson, Principles of continuous choice and continuity of functions in formal systems for constructive mathematics, *Annals of Mathematical Logic* **12** (1977) 249–322.

[2] M. Beeson, *Foundations of Constructive Mathematics* (Springer, 1985).

[3] S. Feferman, Constructive theories of functions and classes, in: M. Boffa, D. van Dalen and K. McAloon, eds., *Logic Colloquium '78, Proceedings of the Logic Colloquium at Mons*, (North-Holland, 1979) 159–224.

[4] S. Feferman, Polymorphic typed lambda-calculi in a type-free axiomatic framework, Logic and computation (Pittsburgh, 1987), *Contemporary Mathematics* **106** (1987) 101–136.

[5] S. Feferman and G. Jäger, Systems of explicit mathematics with non-constructive μ-operator, Part II, *Annals of Pure and Applied Logic* **79** (1996) 37–52.

[6] S. Hayashi and H. Nakano, **PX**: *A Computational Logic* (MIT Press, Cambridge, 1988).

[7] S. Hayashi, Singleton, Union and Intersection Types for Program Extraction, *Information and Computation* **109** (1994) 174–210.

[8] S. Kobayashi and M. Tatsuta, Realizability interpretation of generalized inductive definitions, *Theoretical Computer Science* **131** (1) (1994) 121–138.

[9] B. Nordström, K. Petersson and J.M. Smith, *Programming in Martin-Löf's Type Theory* (Oxford University Press, Oxford, 1990).

[10] C. Paulin-Mohring, Extracting F_ω's programs from proofs in the Calculus of Constructions, *Proceedings of Symposium on Principles of Programming Languages* (1989) 89–104.

[11] M. Tatsuta, Program Synthesis Using Realizability, *Theoretical Computer Science* **90** (1991) 309–353.

[12] M. Tatsuta, **q**-realizability without Doubling Variables for Feferman's Theory, Manuscript (1993).

[13] M. Tatsuta, Realizability interpretation of coinductive definitions and program synthesis with streams, *Theoretical Computer Science* **122** (1994) 119–136.

[14] M. Tatsuta, Realizability of Monotone Coinductive Definitions and Its Application to Program Synthesis, Proceedings of Fourth International Conference on Mathematics of Program Construction, *LNCS* (1998).