①

Reflection Mechanism in Constructive Programming

Yukiyoshi Kameyama

April, 1996

Abstract

This thesis studies the role of the reflection mechanism in Constructive Programming.

Constructive Programming is a method of program development based on constructive logic in which correct programs are automatically extracted from proofs of given specifications. Recently it has been widely accepted that the reflection mechanism is quite useful in Constructive Programming.

$\mathcal{RPT}$, *Reflective Proof Theory*, is a constructive logic proposed by Sato[34]. Since the reflection mechanism is built-in in $\mathcal{RPT}$, it can be a suitable basis for our study. In Chapter 2, we first propose a formal system of $\mathcal{RPT}$. We then give several theorems in the formal system, which show the expressive power of the reflection mechanism of $\mathcal{RPT}$. Finally, we study metamathematical properties of the formal system. In particular, the strong normalization theorem and the consistency are proved for our formal system without inductive definitions.

In Chapter 3, we describe our Constructive Programming System based on the formal system given in Chapter 2. The system is implemented by the programming language Λ, which is at the same time the object language of $\mathcal{RPT}$. We give an overview of our Constructive Programming System. As a substantial example of proof development using our system, we demonstrate a mechanized proof of the Church-Rosser property of the programming language Λ. We also present a concrete example of Constructive Programming based on our system, and present a method to eliminate redundant parts from a program.

In Chapter 4, we will study yet stronger reflection mechanism. Although the reflection mechanism in $\mathcal{RPT}$ is quite useful, we cannot re-define a modified provability relation internally. The re-definition of the provability relation is the key to eliminate redundant parts in extracted programs. To solve this problem, we propose the mechanism of *half-monotone* inductive definitions. A half-monotone inductive definition is an extension of the conventional monotone inductive definition so that we can define the provability relation naturally. We give a theory and a realizability interpretation of the half-monotone inductive definitions. We also interpret several theories such as Martin-Löf's type theory and the Logical Theory of Constructions using this mechanism. We also apply the mechanism to the provability relations and show a method of program refinement.

In Chapter 5, we turn our attention to programming languages. The program-

ming language Λ given in Chapter 2 is a purely functional one in the sense that there are no side-effects. It is an interesting research problem to introduce imperative features into our language. The programming language Λ! (pronounced "lambda bang") was proposed by Sato[36] as an extension of Λ in which the assignment and the `while` statements are introduced. We give some conservativeness results on Λ! in Chapter 5.

In Chapter 6, we give concluding remarks of the thesis.

Acknowledgements

I would like to express my heartful thanks to Professor Masahiko Sato of Kyoto University for encouragements and continual supports. I would like to thank Professor Takayasu Ito, Professor Taiichi Yuasa, Professor Makoto Tatsuta, Professor Peter Dybjer, Dr. Carolyn Talcott, Mr. Yasuyuki Tsukada, Mr. Atsuhiko Yamanaka for helpful comments and pointing out errors in earlier drafts.

Finally, I wish to thank my wife Kaori for patience and encouragements during the period of writing this thesis.

Contents

Chapter 1

Introduction

1.1 Backgrounds

Constructive Logic and Constructive Programming

Constructive Programming is a method of program development based on constructive logic in which program extraction from logical specifications and the correctness of programs are guaranteed[18, 10, 11, 21, 33, 30, 22].

A constructive logic is a logic used for reasoning in constructive mathematics[8, 7]. It is not a single logic; rather, it can be any logic which allows the BHK(Brouwer-Heyting-Kolmogorov)-interpretation[47]: The BHK-interpretation is that, a formula holds if it has a proof, and a proof of each formula is defined depending on the outermost logical connective. For example,

> (The case $\vee$) A proof of a disjunctive formula $A \vee B$ is either a proof of A or a proof of B.
>
> (The case $\exists$) A proof of an existentially quantified formula $\exists x.A(x)$ is (a combination of) a term t and a proof of $A(t)$.

As an example, the first-order (or higher-order) intuitionistic logic is a constructive logic while the classical logic cannot be constructive, since the law of the excluded middle $A \vee \neg A$ holds although we cannot decide which of A and $\neg A$ holds in general.

In a constructive logic, given a proof of $\forall x.\exists y.\ A(x, y)$ we can extract a program f and at the same time a proof of $\forall x.A(x, f(x))$ from the proof. If we regard x, y and $A(x, y)$ as the input, the output, and the input-output relation, respectively, then the formula $\forall x.\exists y.A(x, y)$ means that, "for any input x, there exists an output y such that the input-output relation is satisfied". Hence, the formula can be regarded as specification. By the fact above, we have the proof of $\forall x.A(x, f(x))$ which ensures that the program f satisfies the input-output relation, hence the program f is correct with respect to the specification. In other words, we can synthesize a *verified program* from a proof of the specification formula. This is the principle of Constructive Programming.

There is another way of Constructive Programming which is based on type theories rather than logical systems. By the well known Curry-Howard isomorphism[24], a specification formula and its proof (in a constructive logic) correspond to a type and a term of the type (in a type theory). Hence, in a type theory we can do Constructive Programming as well: a specification is written in the form of a type $\Pi x \in S.\Sigma y \in T.A(x,y)$ where Π and Σ are the product and the sum type-constructors, S, T, and $A(x,y)$ are types. If we can give a proof p of this type, namely we have $p : \Pi x \in S.\Sigma y \in T.A(x,y)$, then we can extract two terms f and q such that $q : \Pi x \in S.A(x, f(x))$ holds. Hence, we can extract a correct program f with respect to the specification.

In both ways, the paradigm of Constructive Programming is to write a proof instead of a program, and then synthesize a correct program. Both ways of Constructive Programming have been intensively studied recently: Feferman's T_0[18], Hayashi's **PX**[21], Sato's $\mathcal{SST}$[33] for untyped logics based on the intuitionistic logic, and Martin-Löf's type theory[29], Coquand and Huet's Calculus of Constructions[11] for type theories.

Since the aim of Constructive Programming is not only to pursue logical principles, but also to apply principles to program development, we need to extend or modify the basic systems (logical systems or type theories) so that we can express various kinds of data types and algorithms, and that we can reason about these structures. Among many theories, Sato's $\mathcal{RPT}$, Reflective Proof Theory[34] is yet another constructive logic which is intended to be a basis for Constructive Programming. $\mathcal{RPT}$ is based on an untyped first-order logic, but it also has a feature of type theories in that it explicitly has a term which represents a proof. Therefore we may regard $\mathcal{RPT}$ as a mixture of the two ways. It is interesting to study Constructive Programming based on $\mathcal{RPT}$.

Constructive Programming System

In Constructive Programming, a programmer does not actually write a program directly; instead he or she writes a proof of a specification formula. The proof must be correctly constructed, otherwise, the correctness of the extracted program is not guaranteed.

We therefore need a mechanical proof-checker for our proofs. A *Constructive Programming System* is a computer software which supports men to develop a proof in the paradigm of Constructive Programming. However, a Constructive Programming system can do more; it may support for inputting long, complex formulas, proving a certain class of theorems automatically, extracting a program from a proof, and executing the extracted program.

Corresponding to constructive logics and type theories, there have been designed several Constructive Programming Systems; Hayashi and Nakano's **PX** system[21], Nuprl system[10] in Cornell university, Coq system[14] in INRIA, and Pollack's LEGO system[28]. Each system has its own characteristics. One demerit of theses systems is that they are implemented by large-fledged programming languages while the object languages for their logics/type-theories are quite simple. There is

a big gap between the implementation languages (metalanguages) and the object languages, hence these systems cannot reason about the systems themselves.

Reflection Mechanism

The reflection principle in logic L is to relate a formula and its formalized form:

$$Provable_L("A") \supset A$$

where "A" represents a formalized representation (such as encoding by Gödel numbers) of A, and the predicate $Provable_L(\_)$ is a predicate which internalizes the provability of the logic L.

This "principle" is not really a theorem in L since it does not hold usually. Adding the principle to the logic L results in a stronger logic L_1. We can consider $Provable_{L_1}("A")$, and we have the reflection principle again. Adding it to L_1 results in a stronger system L_2, and this process generates an infinite hierarchy of logical systems.

The reflection mechanism is useful in developing constructive proofs. For instance, let us consider the following statement (in a usual propositional logic):

If A is a formula, then $A \supset A$ is provable.

This statement is a metaformula; namely, it is outside of the logic. Nevertheless we know that it is true, and we want to make use of the metatheorem to prove something directly. From the viewpoint of Constructive Programming, we cannot use this metatheorem to develop proofs, since there is no justification for using this metatheorem. However, if we have some facility of reflection, namely, if we can internalize the metatheorem above, and we can guarantee the correctness of the internalized metatheorem, then we often have a quick and direct proof. There are many other examples of such a metatheorem.

Another example is the principle of Constructive Programming in a first-order theory:

From $\vdash \forall x.\exists y.A(x,y)$,
we have $\vdash \forall x.A(x, f(x))$ for some term f

where $\vdash A$ means that A is provable. This statement itself is a metatheorem since we cannot express it as a formula in a first-order theory. We therefore have to prove it outside the logic, hence the correctness of the principle itself cannot be mechanically checked by a proof-checker for the logic. However, if we can internalize and prove this statement, then the theorem becomes internal, and we can mechanically check it. In $\mathcal{RPT}$, we can actually prove a formalized version of the principle above as follows:

$$\vdash_{i+1} (\vdash_i \forall x.\exists y.A(x,y)) \supset \exists f.(\vdash_i \forall x.A(x, f(x)))$$

This expression is just a formalized principle if we ignore the indices i and $i+1$. We will explain the meaning of this kind of expressions in the thesis.

Note that, our approach reflects a *model* of the formalized world as in the example of $Provable_L("A")$ above. In this example, the predicate $Provable_L("A")$ is defined so that it can reflect the provability of the formula A. The axioms and inference rules are not reflected, and the provability is solely reflected. On the contrary, Allen et al[6]'s approach reflects a whole syntax of the world at a lower level in the sense that all the axioms and inference rules are faithfully reflected. These two approaches are quite different. We will take the first approach, since it is more natural from the logical point of view (the reflection mechanism in logic takes this approach), and moreover, we can construct a much simpler theory than the second approach.

Note also that, the reflection treated in this thesis is derived from logic, and does not have direct connection to the computational reflection which originates from Smith's work[39]. We hope that we can find some connection between them in future.

1.2 Goal of this research

Our aim is to realize the paradigm of Constructive Programming. In order to do so, we need to have a suitably designed programming language as well as a suitably formulated constructive logic by which we can reason about properties of a program in the language. We also have to implement a Constructive Programming System which supports proof-development in the logic. Further we have to study how to make an efficient program by Constructive Programming, since naively extracted programs tend to be quite inefficient.

Firstly, we have to formalize a constructive logic which is expressive enough to represent various kinds of data types and algorithms, and at the same time, has the reflection mechanism. Sato's $\mathcal{RPT}$ is one such logic; it has a strong mechanism of inductive definitions as well as a built-in reflection mechanism. In order to use $\mathcal{RPT}$ as our underlying logic, we have to give a formal system which corresponds to $\mathcal{RPT}$.

Secondly, we have to give a Constructive Programming System for (the formal system of) $\mathcal{RPT}$. In order for the system to reason about its properties, the system must be implemented by the object language of $\mathcal{RPT}$. The system also must have a good user-interface so that we can actually work on the system. To demonstrate the effectiveness and the usefulness of our system, we have to construct substantial examples of Constructive Programming.

Thirdly, we have to give a way to improve the efficiency of extracted programs. So far much research has been done on this topic[32, 23]. Since they propose a uniform way of improving programs in a metatheory, the correctness is outside of the system. It is quite valuable if we can formalize their techniques of improvement, and prove the correctness internally, since we can extract the improvement function from the proof of this (internalized) metatheorem. Although $\mathcal{RPT}$ has the reflection mechanism, we cannot use it to re-define the provability relation. We therefore have to extend the mechanism so that we can define the provability relation internally.

Fourthly, we have to study the programming language again. The programming language in $\mathcal{RPT}$ is a purely functional language Λ. Λ is expressive enough to implement our Constructive Programming System. However, to extend the paradigm of Constructive Programming to real programming worlds, we will have to treat imperative programming languages such as FORTRAN and C. Sato also proposed a new version of Λ, which is a purely functional language with the assignment and the `while` statements. We have to analyze the properties of the new Λ for Constructive Programming.

1.3 Outline of the thesis

$\mathcal{RPT}$, *Reflective Proof Theory* is a constructive logic proposed by Sato[34]. Since the reflection mechanism is built-in, it is a suitable theory for our study.

In Chapter 2, we first propose a formal system of $\mathcal{RPT}$. As in Martin-Löf's type theory, $\mathcal{RPT}$ was given by Sato[34] as a semantical theory in order to give foundation of mathematics. By a semantical theory, we mean that every concept is expressed *semantically*, and by no means a formal system. Since the semantical theory is expressed in an intelligible way, it is in most cases easy for men to determine whether something is true in $\mathcal{RPT}$ or not. However, we need a formal system of $\mathcal{RPT}$ in order to construct a computer implementation. Moreover, if we have a formal system, then we can compare the system to other systems in a rigid way.

We then give several theorems in the formal system, which shows the expressive power of the reflection mechanism of $\mathcal{RPT}$. Finally, we study metamathematical properties of the formal system, in particular the strong normalization theorem for a subsystem of our formal system. As a corollary, we have the consistency of our formal system without inductive definitions.

In Chapter 3, we describe our Constructive Programming System based on our formal system. The system is implemented by the programming language Λ, which is at the same time the object language of our formal system. We give an overview of our Constructive Programming System. As a substantial example of our system, we demonstrate a mechanized proof of the Church-Rosser property of our programming language Λ. We also present a concrete example of Constructive Programming based on our system, and present a method to eliminate redundant parts from a naive program.

In Chapter 4, we will study yet stronger reflection mechanism. Although the reflection mechanism in $\mathcal{RPT}$ is quite useful, we cannot re-define a modified provability relation internally. The re-definition of the provability relation is the key to eliminate redundant parts of extracted programs. To solve this problem, we propose the mechanism of *half-monotone* inductive definitions. A half-monotone inductive definition is an extension of the ordinary monotone inductive definition so that we can define the provability relation naturally. We give a theory, a realizability interpretation, and several applications of this mechanism.

In Chapter 5, we turn our attention to programming languages. The programming language Λ given in Chapter 2 was a purely functional one in the sense that there are no side-effects. It is an interesting research problem to introduce imperative features into our language. The new version of Λ (called Λ`!` in this thesis) is an extension of Λ in which the assignment and the `while` statements are introduced by Sato[36]. We will give some conservativeness results on Λ`!` in Chapter 5.

In Chapter 6, we give concluding remarks of the thesis.

Chapter 2

Formalizing Reflective Proof Theory

Reflective Proof Theory ($\mathcal{RPT}$ in short) is a logical system which is aimed at a basis for Constructive Programming introduced by Sato[34].

As Martin-Löf's type theory[29], $\mathcal{RPT}$ was given as a semantical theory in order to give foundation of mathematics. By a semantical theory, we mean that every concept is expressed *semantically*, and by no means a formal system. The semantical theory is explained so that it is easy for men to determine whether a judgement is true in $\mathcal{RPT}$ or not. However we need a formal system of $\mathcal{RPT}$ in order to implement a proof-development system such as a proof-checker and a prover. Moreover, if we have a formal system, we can compare the system to other systems in a rigid way.

In this chapter we first give RPT, a formal system for $\mathcal{RPT}$. (In the following we will write RPT for the formal system, and $\mathcal{RPT}$ for the semantical theory presented in [34].) Our formal system is constructed in such a way that $\mathcal{RPT}$ is a model of RPT. We then give several theorems which demonstrate the expressive power of RPT.

We also give several metamathematical studies on the system RPT. Among them we will prove the strong normalization property for a subsystem of RPT, by which we can derive several important properties on RPT.

2.1 Semantical Theory $\mathcal{RPT}$

In this section, we briefly describe Sato's *Reflective Proof Theory* ($\mathcal{RPT}$, in short) in [34].

$\mathcal{RPT}$ is an extension of Aczel's Frege structures [2] in the direction of Constructive Programming.

A Frege structure is constructed from an arbitrary model of the λ calculus. Let M be a domain of the model. By an appropriate encoding, we can assume that constants $\dot{\bot}, \dot{\wedge}, \dot{\vee}, \dot{\supset}, \dot{\forall}$, and $\dot{\exists}$ are included in M. These constants correspond to the logical symbols $\bot, \wedge, \vee, \supset, \forall$, and $\exists$.

Then, two subsets P and Q of M (where $P \supseteq Q$) are defined. The intuitive meaning of P is the set of propositions (represented as λ-terms), and that of Q is the set of true propositions (represented as λ-terms).

- $\dot{\bot} \in P$.
- $\dot{\bot} \notin Q$.
- $\dot{\wedge}ab \in P$ if and only if $a \in P$ and $b \in P$.
- $\dot{\wedge}ab \in Q$ if and only if $a \in Q$ and $b \in Q$.
- $\dot{\vee}ab \in P$ if and only if $a \in P$ and $b \in P$.
- $\dot{\vee}ab \in Q$ if and only if either (i) $a \in Q$ and $b \in P$, or (ii) $a \in P$ and $b \in Q$.
- $\dot{\supset}ab \in P$ if and only if $a \in P$ and if $a \in Q$, then $b \in P$.
- $\dot{\supset}ab \in Q$ if and only if either (i) $a \in P$ and $a \notin Q$, or (ii) $a \in P$ and $b \in Q$.
- $\dot{\forall}a \in P$ if and only if $ab \in P$ for any term b.
- $\dot{\forall}a \in Q$ if and only if $ab \in Q$ for any term b.
- $\dot{\exists}a \in P$ if and only if $ab \in P$ for any term b.
- $\dot{\exists}a \in Q$ if and only if $ab \in Q$ for some term b.

If $a \in P$, then a is called a proposition, and if $a \in Q$, a is called a true proposition.

By regarding a unary propositional function as a set, we can extend the Frege-style set theory (where set-comprehension is allowed) in Frege structures. Aczel used Frege structures to analyze Russell's paradox[2].

Sato proposed $\mathcal{RPT}$ with the following extensions to Frege structures:

Extension of the domain

The terms in $\mathcal{RPT}$ are pure λ-terms enriched with several terms such as pairs (`cons` in Lisp), `if` a `then` b `else` c `fi` and so on. Since these new terms can be represented by pure λ-terms with some encoding, this extension is inessential from the viewpoint of the computational power. However, it is useful from the viewpoint of Constructive Programming, since terms in $\mathcal{RPT}$ are programs.

Explicit proof

Frege structures have two basic judgements "a is a proposition", and "a is a true proposition".

$\mathcal{RPT}$ also has two judgements, one of which is exactly the same as the first one. The other one is of the form "p is a proof of a proposition a", which is an extension of the second one in that the proof of a is explicitly shown. This extension is quite natural if we take the intuitionistic dogma where a proposition is true if and only if it has a proof.

Formally, the judgements "a is proposition" and "p is a proof of a" are represented as $\models_i a$ and $p \vdash_i a$, respectively. Here the suffix i is the *level* in the reflective tower explained below. Note that the symbols $\models$ and $\vdash$ usually mean *truth in model* and *provability in a formal system* which are completely different in $\mathcal{RPT}$. The judgement $a \vdash_i p$ in $\mathcal{RPT}$ corresponds to the judgement $a : p$ in type theories, and "a is a realizer of the formula p" in realizability interpretation for type-free logics.

Reflective tower

In a usual logical system, "if p is a proposition (formula), then $p \supset p$ is a true proposition (formula)" is a true statement, but we cannot represent itself as a proposition (formula). This kind of metapropositions (schemata) is quite useful in its expressiveness. However, if we would naively introduce metapropositions, namely, if we would simply regard metapropositions as propositions, we would fall into Russell's paradox, and the whole system would be inconsistent. In order to avoid the inconsistency, $\mathcal{RPT}$ introduces the level for each proposition, and regards a metaproposition of level i as a level $i+1$ proposition. Therefore we have a family of sets of propositions, each of which is indexed by ordinal numbers. Moreover, if $i < j$, the set indexed by i is a subset of the set indexed by j. Hence the family is an increasing sequence of sets of propositions. We call this family a *reflective tower*. By virtue of the reflective tower, we can successfully construct a consistent theory in which metapropositions can be expressed.

The reflective tower has a considerable benefit in Constructive Programming. Let us take an example of this benefit; For a logical system to be used for Constructive Programming, it must satisfy the so-called term existence property as follows:

Proposition 1 (Term Existence Property) *If we can prove $\exists x.A(x)$, then we can effectively obtain a term t and a proof of $A(t)$.* □

In order to prove this property for a first-order logic such as Beeson's EON [7], one may use techniques such as the realizability interpretation, normalization, and so on. All these techniques need reasoning in a metalevel (outside of the logical system itself). On the other hand, we can internally formalize the term existence property in $\mathcal{RPT}$.

Proposition 2 *In $\mathcal{RPT}$, we can prove the following judgement for some term a (if $f(b)$ is always a proposition for any term b):*

$$a \vdash_{i+1} ((\vdash_i \exists x.f(x)) \supset (\exists z. \vdash_i f(z)))$$

This judgement means that from a proof of the proposition $\exists x.f(x)$, we can (effectively) obtain z with $f(z)$ being true.

Inductive definition

Since recursive data structures are quite important in programming practice, our logical system must have the mechanism of induction definitions (and corresponding induction principles).

$\mathcal{RPT}$ has a general mechanism of inductive definitions. Inductive definition is treated in more depth in Chapter 4.

Sato proposed $\mathcal{RPT}$ by extending Frege structures with the four extensions above.

$\mathcal{RPT}$ resembles Frege structures in that it is a semantical theory, and is not a formal system. In order to implement a Constructive Programming System, we need to formalize $\mathcal{RPT}$. A formalization of $\mathcal{RPT}$ is described in the next section.

2.2 Formal System RPT

This section presents the system RPT, which is a formalization of $\mathcal{RPT}$.

2.2.1 The target of the formalization

The semantical theory $\mathcal{RPT}$ was given in [34].

By Gödel's incompleteness theorem, any formalization of $\mathcal{RPT}$ is incomplete. Hence, our goal in formalizing $\mathcal{RPT}$ is to formalize a large part of $\mathcal{RPT}$.

In the original version of $\mathcal{RPT}$[34], the judgement $p \vdash_i a \wedge b$ does not necessarily imply that p is a pair ($q \vdash_0$ `pair?`$(p) =$ `true` for some q). However, the intended semantical theory of $\mathcal{RPT}$ was that the proof term of $a \wedge b$ should be a pair. (If $p \vdash_i a$ holds, then we say p is a *proof term* of the proposition a.) Hence, we put new conditions on $\mathcal{RPT}$ so that proof terms of $p \wedge q$, $p \,\&\, q$, $p \vee q$, and $\exists p$ must be pairs, and those of $p \supset q$ and $\forall p$ must be functions. In this paper, we assume that $\mathcal{RPT}$ satisfies these conditions.

Compared with $\mathcal{RPT}$, we put the following restrictions to RPT.

1. Restriction on the level of reflective tower

   In $\mathcal{RPT}$, a level in the reflective tower can be an arbitrary ordinal number. There is an example in [34] which uses the level $\omega+1$ where ω is the first infinite

   ordinal number (corresponding to the set of natural numbers). Moreover, there are variables for levels, and they can be quantified by $\forall$ or $\exists$. For instance,

   $$t \ \vdash_\omega \ \forall i < \omega.\forall a.((\models_i a) \supset a \supset a)$$

   is a true judgement if we put $t \equiv \lambda ixayz.z$. This judgement formalizes "for each level less than ω, if a is a proposition, then $a \supset a$ is a true proposition".

   From the viewpoint of real Constructive Programming, we do not need such a high level (ω or a larger ordinal), nor level-variables. We therefore decided to restrict the level to be integer constants. In the following, the metavariables i, j for levels represent integer constants (represented by some terms in RPT through appropriate encoding).

2. Restriction on Inductive definition

   In $\mathcal{RPT}$, a new predicate can be inductively defined under the condition of "strictly positiveness". This condition is, for every sub-proposition of the form $A \supset B$ in the body of the inductive definition, the predicate variable (being defined through this definition) does not appear in A. For example, $X(a) \vee (b = 0 \supset X(c))$ is strictly positive, while $X(a) \supset b = 0$ is not.

   As stated in [34], this condition is semantical, rather than syntactic. Let X be a unary predicate variable, and P be $(X[x] \supset X[x]) \wedge X[x]$. Then $X[x]$ appears in the lefthandside of $\supset$, so P is not strictly positive. Since P is logically equivalent to $X[x]$, which is strictly positive, so is P *semantically*. Hence, we can define a new predicate using P in $\mathcal{RPT}$.

   However, this example is quite artificial and useless, since we can inductively define the same predicate using $X[x]$. Although there might be cases where the *semantical* strictly positiveness can be useful, we believe that the syntactic condition is sufficient for writing specifications for real programs. We therefore restrict inductive definitions for only syntacticly strictly positive cases. This restriction allows us to remove $\models^+$ and $\vdash^+$ which were used to express the (semantic) strictly positiveness in $\mathcal{RPT}$.

3. Introduction of propositions in the form $a\downarrow$

   We often need the proposition "a term a has a value" through inferences. Since the computation in RPT is call-by-need, it is expressed as:

   $$a = \texttt{nil} \ \vee \ a = \texttt{true} \ \vee \ a = \texttt{false} \ \vee \ \texttt{pair?}(a) = \texttt{true} \ \vee \ \texttt{fun?}(a) = \texttt{true}$$

   This proposition is a disjunction of five atomic propositions, and its proof term is a long one (if it exists). On the other hand, since the proof-term can be computed from a (if it exists), we do not need the proof term besides a itself. In this sense, the proof term does not carry any computational meaning.

Following Beeson[7], we introduced a new atomic proposition $a\downarrow$ which is logically equivalent to the proposition above, but the proof term of $a\downarrow$ is some dummy constant.

2.2.2 Terms and their reduction rules

Terms of RPT are defined as follows.

Definition 1 (Term)

$$\begin{array}{rcl} t & ::= & x \\ & | & \texttt{nil} \mid \texttt{null?}(t) \\ & | & \texttt{true} \mid \texttt{true?}(t) \\ & | & \texttt{false} \mid \texttt{false?}(t) \\ & | & \langle t,t\rangle \mid \texttt{car}(t) \mid \texttt{cdr}(t) \mid \texttt{pair?}(t) \\ & | & \lambda x.t \mid t(t) \mid \texttt{fun?}(t) \\ & | & \mu t \\ & | & \texttt{if } t \texttt{ then } t \texttt{ else } t \texttt{ fi} \end{array}$$

where x is a metavariable for variables. □

Terms $\texttt{nil}, \texttt{car}(t)$ and $\texttt{cdr}(t)$ are the same as those in Lisp. We denote pairs, λ-abstraction, application as $\langle t,t\rangle$, $\lambda x.t$, and $t(t)$, respectively. The term μt invokes a recursive call, for example, the following term is a program for the **Append** function in Lisp.

$$\mu(\lambda f.\lambda x.\lambda y.\texttt{if null?}(x)\texttt{ then } y \texttt{ else } \langle\texttt{car}(x), f(\texttt{cdr}(x))(y)\rangle\texttt{ fi})$$

We say two terms are of the same kind if they are defined in the same row in the definition above, and of different kinds otherwise. For instance, $\texttt{nil}$ and $\texttt{null?}(t)$ are of the same kind, while $\texttt{nil}$ and $\texttt{true?}(t)$ are of different kinds. Terms $\texttt{null?}(t)$, $\texttt{true?}(t)$, $\texttt{false?}(t)$, $\texttt{pair?}(t)$, and $\texttt{fun?}(t)$ are called recognizer terms, and are used for recognizing the kinds of the arguments.

Bound variables, free variables, and substitution are defined as usual. Terms which do not have free variables are called closed terms. The expression $a_{x_1,\dots,x_n}[b_1,\dots,b_n]$ is the result of simultaneous substitution of $b_1,\dots,b_n$ for $x_1,\dots,x_n$ in the term a. $FV(a)$ is the set of free variables in a.

We also use the following abbreviations:

$$\begin{array}{rcl} ab & \triangleq & a(b) \\ a(b_1,\dots,b_n) & \triangleq & a(b_1)\cdots(b_n) \\ \langle\,\rangle & \triangleq & \texttt{nil} \end{array}$$

$$\begin{array}{rcl} \langle a\rangle & \triangleq & \langle a,\texttt{nil}\rangle \\ \langle a\,b\rangle & \triangleq & \langle a,\langle b,\texttt{nil}\rangle\rangle \\ \langle a\,b\,c\rangle & \triangleq & \langle a,\langle b,\langle c,\texttt{nil}\rangle\rangle\rangle \\ \lambda xy.a & \triangleq & \lambda x.\lambda y.a \\ \lambda xyz.a & \triangleq & \lambda x.\lambda y.\lambda z.a \end{array}$$

We then define canonical terms and normal terms.

Definition 2 (Canonical term)

$$\begin{array}{rcl} c & ::= & \texttt{nil} \mid \texttt{true} \mid \texttt{false} \\ & | & \langle t,t\rangle \mid \lambda x.t \end{array}$$

Definition 3 (Normal term)

$$\begin{array}{rcl} n & ::= & x \\ & | & \texttt{nil} \mid \texttt{true} \mid \texttt{false} \\ & | & \langle n,n\rangle \mid \lambda x.n \end{array}$$

When a normal (canonical) term b is obtained by evaluating a, then we say b is a normal (canonical) form of the term a.

The evaluation mechanism of RPT is essentially call-by-name. For instance, when we evaluate $\texttt{pair?}(a)$, the argument a is evaluated to a canonical term, and not necessarily to a normal term. Hence, the canonical terms are important in the evaluation of RPT.

We encode the natural numbers as follows:

$$0 \triangleq \texttt{nil}$$

$$i+1 \triangleq \langle\texttt{nil}, i\rangle$$

Propositions and judgements are also terms via the following encoding.

$$\begin{array}{rcl} a=b & \triangleq & \langle 1\,a\,b\rangle \\ a[b] & \triangleq & \langle 2\,a\,b\rangle \\ \models_i b & \triangleq & \langle 3\,i\,b\rangle \\ a\vdash_i b & \triangleq & \langle 5\,i\,a\,b\rangle \\ a\downarrow & \triangleq & \langle 6\,a\rangle \\ a\wedge b & \triangleq & \langle 7\,a\,b\rangle \\ a\,\&\,b & \triangleq & \langle 8\,a\,b\rangle \end{array}$$

$$\begin{aligned} a \vee b &\triangleq \langle 9\, a\, b\rangle \\ a \supset b &\triangleq \langle 10\, a\, b\rangle \\ \forall a &\triangleq \langle 11\, a\rangle \\ \exists a &\triangleq \langle 12\, a\rangle \end{aligned}$$

The expression $a[b]$ represents an atomic proposition with a being a user-defined predicate. The meaning of other expressions will be explained later. We abbreviate $\forall(\lambda x.b)$, $\exists(\lambda x.b)$, and $a \supset$ `false` as $\forall x.b, \exists x.b$, and $\neg a$, respectively.

We assume that $\wedge, \&, \vee, \supset$ associate to the right; for example, $a \supset b \supset c$ is an abbreviation of $a \supset (b \supset c)$.

Definition 4 (Evaluation of terms) *For two terms a and b, we define $a \to b$ as in the table below.*

Reduction Rule	Condition
$\#(a) \to$ `true`	$\#(a)$ is a recognizer term, a is canonical, and $\#(a)$ and a are of the same kind
$\#(a) \to$ `false`	$\#(a)$ is a recognizer term, a is canonical, and $\#(a)$ and a are of different kinds
`car`$(\langle a, b\rangle) \to a$	
`cdr`$(\langle a, b\rangle) \to b$	
$(\lambda x.a)b \to a_x[b]$	
$\lambda x.(\lambda y.a)x \to \lambda y.a$	$x \notin FV(a)$
$\mu a \to a(\mu a)$	
`if true then` a `else` b `fi` $\to a$	
`if false then` a `else` b `fi` $\to b$	
$C[b] \to C[c]$	$b \to c$

In the last rule, $C[\,]$ is a context in a usual sense.

The binary relation $\to$ is so-called 1-step reduction. We define two relations $\to^*$ and $\approx$ as the reflexive, transitive closure of $\to$, and the least equivalence relation which subsumes $\to^*$, respectively.

The evaluation does not always terminate as in the usual untyped λ-calculus. We define Ω to be $(\lambda x.xx)(\lambda x.xx)$ as an example of non-terminating terms.

The programming language Λ is thus defined. The Church-Rosser Property (confluency) of Λ was proved in [34]. In Section 3.3, we will formally prove this property using our system.

The evaluation in Λ is nondeterministic. The call-by-name evaluation strategy is a normalizing strategy. We therefore implemented this strategy on a computer. In the next Chapter, we use the programming language Λ to implement our Constructive Programming System. Hence, Λ is a target language of our reasoning in RPT, and at the same time, our implementation language.

2.2.3 Judgement and inference rules

As in Martin-Löf's type theory, we have judgements as components of proofs (proof-figures) in RPT.

Definition 5 (Judgement) *Let a and p be terms, and i be a natural number (encoded by terms). Then the following two forms are judgements of RPT:*

$$\models_i p$$

$$a \vdash_i p$$

The first judgement means that p is a level-i proposition, and the second one means that a is a proof of a level-i proposition p. For the judgement $a \vdash_i p$, we call a as the proof term of p. In the following, we use metavariables $a, b, c, p, q, s, t, \cdots$ for terms, i, j for natural numbers, J, J_i for judgements, f, g, x, y, z for variables.

Inference rules of RPT is written in the natural deduction style. They are classified into the following categories.

1. Rules for levels
2. Rules for equality and canonical terms
3. Rules for propositions
4. Rules for proof terms (truth)
5. Rules for predicates

We will describe each category in detail.

Rules for levels

$$\boxed{\begin{array}{c} \dfrac{\models_j p}{\models_i p}\ (i > j) \qquad \dfrac{a \vdash_j p}{a \vdash_i p}\ (i > j) \\[2ex] \dfrac{a \vdash_i p \quad \models_j p}{a \vdash_j p}\ (i > j) \end{array}}$$

From the first two rules, if some judgement holds in a certain level, then it always holds in a higher level. From the last rule, if the judgement $a \vdash_i p$ holds, then it already holds at the level where p becomes a proposition. In other words, whether a proposition p is true or not (has a proof or not) is determined at the lowest level where p becomes a proposition.

Rules for equality and canonical terms

$$\frac{}{0 \vdash_i a = b} \text{ (where } a \approx b\text{)}$$

$$\frac{}{\lambda x.0 \vdash_i \neg(a = b)} \text{ (with the side condition below)}$$

$$\frac{a \vdash_i b_x[d] \quad c \vdash_i d = e}{a \vdash_i b_x[e]} \qquad \frac{a_x[d] \vdash_i b \quad c \vdash_i d = e}{a_x[e] \vdash_i b}$$

$$\frac{a \vdash_i b = c}{0 \vdash_i a = 0} \qquad \frac{\models_i a_x[d] \quad c \vdash_i d = e}{\models_i a_x[e]}$$

The side condition of the second rule is that a and b are canonical terms of different kinds.

The third and fourth rules (in the third row) are called the equal-right and the equal-left inference rules.

$$\frac{p \vdash_i \langle a, b\rangle = \langle c, d\rangle}{p \vdash_i a = c} \qquad \frac{p \vdash_i \langle a, b\rangle = \langle c, d\rangle}{p \vdash_i b = d}$$

$$\frac{p \vdash_i \texttt{null?}(a) = \texttt{true}}{p \vdash_i a = \texttt{nil}} \qquad \frac{p \vdash_i \texttt{true?}(a) = \texttt{true}}{p \vdash_i a = \texttt{true}}$$

$$\frac{p \vdash_i \texttt{false?}(a) = \texttt{true}}{p \vdash_i a = \texttt{false}} \qquad \frac{p \vdash_i \texttt{pair?}(a) = \texttt{true}}{p \vdash_i a = \langle \texttt{car}(a), \texttt{cdr}(a)\rangle}$$

$$\frac{p \vdash_i \texttt{fun?}(a) = \texttt{true}}{p \vdash_i \#(a) = \texttt{false}} \text{ (with the side condition below)}$$

$$\frac{p \vdash_i \texttt{fun?}(a) = \texttt{true}}{p \vdash_i a \downarrow}$$

The side condition of the second last rule is that $\#(a)$ is a recognizer term not of the form $\texttt{fun?}(a)$.

$$\frac{p \vdash_i a}{0 \vdash_i p \downarrow} \qquad \frac{p \vdash_i a \downarrow}{0 \vdash_i p = 0}$$

$$\frac{}{0 \vdash_i a \downarrow} \text{ (where } a \text{ is canonical)}$$

$$\frac{p \vdash_i \#(a) \downarrow}{p \vdash_i a \downarrow} \text{ (where } \#(a) \text{ is a recognizer term)}$$

$$\frac{p \vdash_i \texttt{car}(a) \downarrow}{p \vdash_i \texttt{pair?}(a) = \texttt{true}} \qquad \frac{p \vdash_i \texttt{cdr}(a) \downarrow}{p \vdash_i \texttt{pair?}(a) = \texttt{true}}$$

$$\frac{p \vdash_i a(b) \downarrow}{p \vdash_i \texttt{fun?}(a) = \texttt{true}}$$

$$\frac{p \vdash_i \texttt{if } a \texttt{ then } b \texttt{ else } c \texttt{ fi} \downarrow}{\langle a, 0\rangle \vdash_i a = \texttt{true} \lor a = \texttt{false}}$$

$$\frac{p \vdash_i a \downarrow}{D(a) \vdash_i a = \texttt{nil} \lor a = \texttt{true} \lor a = \texttt{false} \lor \texttt{pair?}(a) = \texttt{true} \lor \texttt{fun?}(a) = \texttt{true}}$$

In the last rule, $D(a)$ is defined as follows:

$$\begin{array}{rcl}
D(a) & \triangleq & \texttt{if null?}(a) \texttt{ then } \langle \texttt{true}, 0\rangle \\
& & \texttt{else if true?}(a) \texttt{ then } c_1 \\
& & \texttt{else if false?}(a) \texttt{ then } c_2 \\
& & \texttt{else if pair?}(a) \texttt{ then } c_3 \\
& & \texttt{else } c_4 \\
& & \texttt{fi fi fi fi} \\
c_1 & \triangleq & \langle \texttt{false}, \langle \texttt{true}, 0\rangle\rangle \\
c_2 & \triangleq & \langle \texttt{false}, \langle \texttt{false}, \langle \texttt{true}, 0\rangle\rangle\rangle \\
c_3 & \triangleq & \langle \texttt{false}, \langle \texttt{false}, \langle \texttt{false}, \langle \texttt{true}, 0\rangle\rangle\rangle\rangle \\
c_4 & \triangleq & \langle \texttt{false}, \langle \texttt{false}, \langle \texttt{false}, \langle \texttt{false}, 0\rangle\rangle\rangle\rangle
\end{array}$$

Rules for propositions

Most propositions in RPT are the same as formulas in the usual first-order logic; we have $\texttt{true}$, $\texttt{false}$, $a \downarrow$, $a = b$, $\models_i p$, and $a \vdash_i p$ as atomic propositions, and $\land, \&, \lor, \supset, \forall$ and $\exists$ as logical connectives. The different points are two points:

- The judgements $\models_i p$ and $a \vdash_i p$ can again be propositions at the level $i + 1$. In this case p is an arbitrary term, and is not necessarily a proposition.
- If p is a proposition which does not have a proof-term (that is, p is a false proposition), then $p \supset q$ and $p \,\&\, q$ are propositions even if q is not a proposition.

The proposition $p \,\&\, q$ is *conditional conjunction.* It has a similar meaning as the usual conjunction, but it can be a proposition for more cases than the usual one. In order for $p \wedge q$ to be a proposition, p and q must be propositions. On the other hand, in order for $p \,\&\, q$ to be a proposition, p must be a proposition, and q must be a proposition if p is true. A concrete example of & is given in Section 2.3.2.

$$
\begin{array}{cc}
\dfrac{}{\models_i a = b} & \dfrac{}{\models_i a \downarrow} \\[2ex]
\dfrac{}{\models_i \texttt{true}} & \dfrac{}{\models_i \texttt{false}} \\[2ex]
\dfrac{}{\models_i (\models_j p)}\,(i > j) & \dfrac{}{\models_i (p \vdash_j q)}\,(i > j)
\end{array}
$$

$$
\begin{array}{ccc}
\dfrac{\models_i p \quad \models_i q}{\models_i p \wedge q} & \dfrac{\models_i p \wedge q}{\models_i p} & \dfrac{\models_i p \wedge q}{\models_i q} \\[2ex]
\dfrac{\models_i p \quad \models_i q}{\models_i p \vee q} & \dfrac{\models_i p \vee q}{\models_i p} & \dfrac{\models_i p \vee q}{\models_i q} \\[2ex]
\multicolumn{2}{c}{\dfrac{\models_i p \quad \begin{array}{c}[x \vdash_i p] \\ \models_i q\end{array}}{\models_i p \supset q}\,(x \notin FV(q))} & \dfrac{\models_i p \supset q}{\models_i p} \\[2ex]
\dfrac{\models_i p \supset q \quad a \vdash_i p}{\models_i q} & \multicolumn{2}{c}{\dfrac{\models_i p \quad \begin{array}{c}[x \vdash_i p] \\ \models_i q\end{array}}{\models_i p \,\&\, q}\,(x \notin FV(q))} \\[2ex]
\dfrac{\models_i p \,\&\, q}{\models_i p} & \multicolumn{2}{c}{\dfrac{\models_i p \,\&\, q \quad a \vdash_i p}{\models_i q}} \\[2ex]
\multicolumn{2}{c}{\dfrac{\models_i p(x)}{\models_i \forall p}\,(x \text{ is an eigen variable})} & \dfrac{\models_i \forall p}{\models_i p(t)} \\[2ex]
\multicolumn{2}{c}{\dfrac{\models_i p(x)}{\models_i \exists p}\,(x \text{ is an eigen variable})} & \dfrac{\models_i \exists p}{\models_i p(t)}
\end{array}
$$

$$
\dfrac{a \vdash_i (\models_j p)}{\models_j p}\,(i > j) \qquad \dfrac{a \vdash_i (\models_j p)}{0 \vdash_i a = 0}\,(i > j)
$$

Rules for proof terms (truth)

Here we assume $i > j$.

$$
\begin{array}{ccc}
\dfrac{}{0 \vdash_i \texttt{true}} & \dfrac{a \vdash_i \texttt{true}}{0 \vdash_i a = 0} & \dfrac{a \vdash_i \texttt{false}}{a \vdash_i p} \\[2ex]
 & \dfrac{\models_j p}{0 \vdash_i (\models_j p)} & \\[2ex]
\multicolumn{3}{c}{\dfrac{a \vdash_j p}{a \vdash_i (a \vdash_j p)} \qquad \dfrac{a \vdash_i (b \vdash_j p)}{a \vdash_j p}}
\end{array}
$$

These rules are called the `true`-intro, the `true`-elim, the `false`-elim, the prop-intro, the level-up, and the level-down inference rules, respectively.

$$
\begin{array}{cc}
\dfrac{a \vdash_i p \quad b \vdash_i q}{(a, b) \vdash_i p \wedge q} & \dfrac{a \vdash_i p \wedge q}{0 \vdash_i \texttt{pair?}(a) = \texttt{true}} \\[2ex]
\dfrac{a \vdash_i p \wedge q}{\texttt{car}(a) \vdash_i p} & \dfrac{a \vdash_i p \wedge q}{\texttt{cdr}(a) \vdash_i q} \\[2ex]
\dfrac{a \vdash_i p \quad \models_i q}{\langle \texttt{true}, a \rangle \vdash_i p \vee q} & \dfrac{\models_i p \quad b \vdash_i q}{\langle \texttt{false}, b \rangle \vdash_i p \vee q} \\[2ex]
\dfrac{a \vdash_i p \vee q}{0 \vdash_i \texttt{pair?}(a) = \texttt{true}} & \dfrac{a \vdash_i p \vee q \quad \begin{array}{c}[P \quad x \vdash_i p] \\ \vdots \\ b \vdash_i r\end{array} \quad \begin{array}{c}[Q \quad x \vdash_i q] \\ \vdots \\ c \vdash_i r\end{array}}{D \vdash_i r}\,(x \text{ is an eigen variable})
\end{array}
$$

These rules are called the $\wedge$-intro, the $\wedge$-elim-0, the $\wedge$-elim-1, the $\wedge$-elim-2, the $\vee$-intro-1, the $\vee$-intro-2, the $\vee$-elim-0, and the $\vee$-elim-1 inference rules, respectively. In the last rule, we used the following definitions:

$$
\begin{aligned}
P &\triangleq 0 \vdash_i \texttt{car}(a) = \texttt{true} \\
Q &\triangleq 0 \vdash_i \texttt{car}(a) = \texttt{false} \\
D &\triangleq (\texttt{if car}(a) \texttt{ then } \lambda x.b \texttt{ else } \lambda x.c \texttt{ fi})(\texttt{cdr}(a))
\end{aligned}
$$

$$\begin{array}{c}[x \vdash_i p] \\ \vdots \\ \dfrac{\models_i p \quad a \vdash_i q}{\lambda x.a \vdash_i p \supset q}\ (x \text{ is an eigen variable})\end{array}$$

$$\frac{a \vdash_i p \supset q}{0 \vdash_i \texttt{fun?}(a) = \texttt{true}} \qquad \frac{a \vdash_i p \supset q \quad b \vdash_i p}{a(b) \vdash_i q}$$

$$\frac{a \vdash_i p \quad b \vdash_i q}{\langle a, b\rangle \vdash_i p \,\&\, q} \qquad \frac{a \vdash_i p \,\&\, q}{0 \vdash_i \texttt{pair?}(a) = \texttt{true}}$$

$$\frac{a \vdash_i p \,\&\, q}{\texttt{car}(a) \vdash_i p} \qquad \frac{a \vdash_i p \,\&\, q}{\texttt{cdr}(a) \vdash_i q}$$

These rules are called the $\supset$-intro, the $\supset$-elim-0, the $\supset$-elim-1, the &-intro, the &-elim-0, the &-elim-1, and the &-elim-2 inference rules, respectively.

$$\frac{a(x) \vdash_i p(x)}{a \vdash_i \forall p}\ (x \text{ is an eigen variable})$$

$$\frac{a \vdash_i \forall p}{0 \vdash_i \texttt{fun?}(a) = \texttt{true}} \qquad \frac{a \vdash_i \forall p}{a(b) \vdash_i p(b)}$$

$$\frac{a \vdash_i p(b)}{\langle b, a\rangle \vdash_i \exists p}$$

$$\frac{a \vdash_i \exists p}{0 \vdash_i \texttt{pair?}(a) = \texttt{true}} \qquad \frac{a \vdash_i \exists p}{\texttt{cdr}(a) \vdash_i p(\texttt{car}(a))}$$

These rules are called the $\forall$-intro, the $\forall$-elim-0, the $\forall$-elim-1, the $\exists$-intro, the $\exists$-elim-0, the $\exists$-elim-1 inference rules, respectively.
Throughout these rules, eigen variables must satisfy the usual eigen variable condition. For instance, for the $\vee$-elim-1 rule, we must have $x \notin FV(p) \cup FV(q) \cup FV(r)$, and moreover, x must not appear in the right two subproofs except the occurrences explicitly shown in this rule.

The rules for propositions determine what is/is not a proof term of a proposition. Since we regard a proposition which has a proof term as a true proposition, it can be said that these rules determine what is/is not a true proposition.

The proof-terms are naturally defined using the Curry-Howard isomorphism. For instance, the proof-term of a conjunctive proposition is the pair of proof-terms of each conjunct (if exists).

Rules for predicates

In RPT, predicates are always defined by inductive definitions.

Let P be $\lambda f.\lambda x.F$. The term F must be strictly positive with respect to f. Namely, for every subterm of the form $p \supset q$ in F, p must not contain f free.

$$\begin{array}{c}[w \vdash_{i+1} (\forall y.\ \models_i f[y])] \\ \vdots \\ \dfrac{\models_i F}{\models_i P[a]}\ (x, f, w \text{ are eigen variables})\end{array}$$

$$\frac{b \vdash_i F_{x,f}[a, P]}{b \vdash_i P[a]} \qquad \frac{b \vdash_i P[a]}{b \vdash_i F_{x,f}[a, P]}$$

Theses rules correspond to the fold and unfold operations.

$$\frac{b \vdash_i \forall x.(F_f[p] \supset p[x])}{ind_P(b) \vdash_i \forall x.(P[x] \supset p[x])}$$

This rule represents the induction principle. Here ind_P is a term which is calculated from P. See [34] for the calculation in detail.

It is straightforward to extend the rules for predicates which have more than one argument.

Examples of predicates

That "p is a unary propositional function" is represented as the proposition $\forall x.\models_i p(x)$ in RPT. In order to define a predicate for this proposition, we need to define PF as follows:

$$\texttt{PF} \triangleq \lambda f.\lambda p.\ \forall x.\models_i p(x)$$

Since PF does not refer to f in the body above, this definition is not really inductive. In this case, the induction principle becomes a trivial rule.

Another example is Nat, the predicate for natural numbers.

$$\texttt{Nat} \triangleq \lambda f.\lambda x.\ x = 0 \ \vee\ \exists y.\ (x = \texttt{suc}(y) \wedge f[y])$$

where $\texttt{suc}(y) \triangleq \langle 0, y\rangle$. We can easily infer $\texttt{Nat}[n]$ is true for each (encoded) natural number n defined before. The proof-term of the induction principle for Nat is:

$$
\begin{aligned}
ind_{\texttt{Nat}} \;\triangleq\; & \lambda r.\mu(\lambda fxq.\ r(x)([\texttt{car}(q). \\
& \quad \texttt{if car}(q)\ \texttt{then cdr}(q) \\
& \qquad \texttt{else}\ [\texttt{cadr}(q).[\texttt{caddr}(q)\ .f(\texttt{cadr}(q))(\texttt{cdddr}(q))]]\ \texttt{fi}))
\end{aligned}
$$

In this definition we used abbreviations in Lisp such as `cadr`. It is easily shown that the induction principle for `Nat` is equivalent to the usual mathematical induction.

2.2.4 $\mathcal{RPT}$ and RPT

As stated before, the target model of our formalization is not really the original $\mathcal{RPT}$ in [34], but $\mathcal{RPT}$ with a little modification according to the intended semantics of $\mathcal{RPT}$ stated at the beginning of this section. We have that this formalization is sound with respect to the modified $\mathcal{RPT}$.

Theorem 1 (Soundness) *$\mathcal{RPT}$ (after modification) is a model of RPT.* □

Corollary 1 (Consistency) *RPT is consistent.* □

$\mathcal{RPT}$ is a model whose domain is the quotient set of the set of terms modulo the equivalence relation $\approx$. Since this is a model of RPT, we have the following corollary.

Corollary 2 *We have $0 \vdash_i a = b$ is provable in RPT if and only if $a \approx b$ holds.* □

This corollary means that the equivalence relation $=$ in RPT faithfully represents the equivalence relation $\approx$ in $\mathcal{RPT}$. The relation $\approx$ is defined via the operational semantics, and is an intensional one. We will introduce an extensional equivalence relation to the set of terms in RPT in Section 3.4.

2.3 Several Theorems in RPT

In this section, we extend the theory RPT, and show several theorems which shows the expressive power of the reflection mechanism of RPT.

2.3.1 S combinator

In a usual propositional calculus, if A, B and C are formulas, then the following formula is provable:

$$F \triangleq (A \supset B \supset C) \supset (A \supset B) \supset A \supset C$$

However, we cannot formalize the whole sentence "if A, B and C are formulas, ..." in a usual logic, since it is a metatheorem. On the contrary, we can formalize and prove the metatheorem in RPT as follows:

$$P_1 \equiv \dfrac{\dfrac{x \vdash_0 A \supset B \supset C \quad z \vdash_0 A}{xz \vdash_0 B \supset C} \quad \dfrac{y \vdash_0 A \supset B \quad z \vdash_0 A}{yz \vdash_0 B}}{xz(yz) \vdash_0 C}$$

$$P_2 \equiv \dfrac{\models_0 A \supset B \supset C \quad \dfrac{\models_0 A \supset B \quad \dfrac{\models_0 A \quad P_1'}{\lambda z.xz(yz) \vdash_0 A \supset C}}{\lambda yz.xz(yz) \vdash_0 (A \supset B) \supset A \supset C}}{S \vdash_0 F}$$

Here P_1' is the proof figure P_1 where three open assumptions were discharged by the $\supset$-introduction rule, and $S \triangleq \lambda xyz.xz(yz)$.

$$P_3 \equiv \dfrac{\begin{array}{c} \dfrac{[u \vdash_1 (\models_0 A)]}{\models_0 A} \quad \dfrac{[w \vdash_1 (\models_0 C)]}{\models_0 C} \quad \dfrac{[v \vdash_1 (\models_0 B)]}{\models_0 B} \\ \vdots\ P_2 \\ \dfrac{S \vdash_0 F}{0 \vdash_1 (S \vdash_0 F)} \\ \vdots \\ \lambda uvw.0 \vdash_1 T \end{array}}{\lambda ABC.\lambda uvw.0 \vdash_1 \forall A.\forall B.\forall C.T}$$

where T is $(\models_0 A) \supset (\models_0 B) \supset (\models_0 C) \supset (S \vdash_0 F)$.

The proof P_3 is a formalized version of the metatheorem.

2.3.2 Set and Russell's paradox

In $\mathcal{RPT}$, we can introduce a set as a unary propositional function, and then we can develop a constructive set theory.

Let p be a term such that, for any term a, $p(a)$ is a level-i proposition. Then p generates the level-i set $\{x \mid p(x) \text{ is true}\}$ by comprehension.

In RPT, this fact can be internally formalized again. Let $\texttt{Set}_i$ and $\in_i$ be the following:

$$\texttt{Set}_i[p] \triangleq \forall x.(\models_i p(x))$$

$$a \in_i p \triangleq \texttt{Set}_i[p]\ \&\ p(a)$$

The proposition $\mathtt{Set}_i[p]$ means that p is the level-i set, and $a \in_i p$ means that p is a set and a is a member of p.

Note that we used & rather than $\wedge$ in the definition of $a \in_i p$. If we would have defined $a \in_i p$ to be $\mathtt{Set}_i[p] \wedge p(a)$, then $a \in_i p$ is not necessarily be a proposition.

As an example of sets, the set of natural numbers, NatSet is defined as follows:

$$\mathtt{NatSet} \stackrel{\triangle}{=} \lambda x.\mathtt{Nat}[x]$$

We can prove that $\mathtt{Set}_0[\mathtt{NatSet}]$ holds and for any natural number n, $n \in_0 \mathtt{NatSet}$ holds.

Russell's paradox in the naive set theory is that, if we can define the set R as $R \stackrel{\triangle}{=} \{x | x \notin x\}$, then we have inconsistency. The modern set theory, namely, ZF set theory avoids this paradox by excluding the set comprehension rule; then we cannot define R as a set in ZF set theory. In RPT, on the other hand, we can define R as a set; we can still avoid inconsistency since our sets are indexed by levels.

Theorem 2 *Let R be $\lambda f.\ \neg(f \in_0 f)$. Then we can prove the following four judgements.*

$$a \vdash_1 \neg\mathtt{Set}_0[R]$$

$$\lambda w.a(\mathtt{car}(w)) \vdash_1 \neg(R \in_0 R)$$

$$\lambda u.0 \vdash_2 \mathtt{Set}_1[R]$$

$$\langle \lambda u.0, \lambda w.a(\mathtt{car}(w))\rangle \vdash_2 R \in_1 R$$

where

$$a \stackrel{\triangle}{=} \lambda u.(\lambda v.\mathtt{cdr}(v)v)\langle u, (\lambda v.\mathtt{cdr}(v)v)\rangle$$

□

R is Russell's set written in the form of a propositional function. The meaning of Theorem 2 is that, R is not a set at level-0, but it is a set at level-1 or higher. We therefore cannot substitute R for f in $\neg(f \in_0 f)$, and we cannot go further. Hence, we can avoid Russell's paradox.

Remark 1 *The term a is a closed proof-term in RPT which cannot be normalized. In other words, the proofs (proof-terms or equivalently proof-figures) in RPT are not necessarily normalizable. This point will be discussed in the next Chapter.*

2.3.3 Use of reflective tower-1

One of the major characteristic points of $\mathcal{RPT}$ is that we can internally express metatheorems by using the reflective tower. In this subsection, we give several examples of the use of the reflective tower.

The Disjunction Property and the Term Existence Property [7] are important properties for constructive logical systems. As stated in Section 2.1, these properties for a first-order logic are proved by metatheoretic arguments. For RPT, we can prove the internally formalized version of these properties.

In the following we abbreviate the proposition $\exists x.\ (x \vdash_i p)$ as $\vdash_i p$ if $x \notin FV(p)$.

Theorem 3 *The disjunction property "if $p \vee q$ is provable, then p or q is provable" is formalized and proved in RPT. Namely, we can prove the following judgement*

$$a \vdash_{i+1} \forall p.\forall q.\ \ (\models_i p) \supset (\models_i q) \supset (\vdash_i p \vee q) \supset (\vdash_i p) \vee (\vdash_i q)$$

for some term a.

Proof. Define a as

$$\lambda pqxyz.\mathtt{if\ cadr}(z)\ \mathtt{then}\ \langle \mathtt{true}, \langle 0, \mathtt{cddr}(z)\rangle\rangle\ \mathtt{else}\ \langle \mathtt{false}, \langle 0, \mathtt{cddr}(z)\rangle\rangle\ \mathtt{fi}.$$

Then we can prove the theorem easily. □

Theorem 4 *The term existence property "if $\exists x.p$ is provable, then we can effectively find a term t and the proof of $p_x[t]$" is formalized and proved in RPT. Namely, we can prove the following judgement*

$$a \vdash_{i+1} \forall f.(\mathtt{PF}[f] \supset (\vdash_i \exists x.f(x)) \supset (\exists z. \vdash_i f(z)))$$

for some term a.

In the formulation of this theorem, we used the predicate $\mathtt{PF}[f]$ to have $f(x)$ as a proposition.

Proof. Define a as $\lambda fpq.\langle \mathtt{cadr}(q), \langle \mathtt{cddr}(q), \mathtt{cddr}(q)\rangle\rangle$. Then we can prove the theorem easily. □

2.3.4 Use of reflective tower-2

We take another example of the use of the reflective tower.

The realizability interpretation is a quite useful technique of program extraction for a wide range of logical systems such as the first-order intuitionistic logic[7]. Let $a\ \mathbf{r}\ P$ mean the term a is a realizer of the formula P. Then $a\ \mathbf{r}\ P$ is semantically similar to the judgement $a \vdash_i P$ in RPT.

Harrop formulas are defined as below in the first-order logic:

Definition 6 (Harrop formula in the first-order logic)

$$\begin{array}{rcl} H & ::= & \texttt{true} \mid \texttt{false} \mid t = t \mid H \wedge H \\ & \mid & P \supset H \mid \forall x.H \end{array}$$

where P is an arbitrary formula. □

If P is a Harrop formula, we have the following well-known result.

Theorem 5 (In the first-order logic) *Let H be a Harrop formula, and $FV(H) \subseteq \{x\}$ holds. Then we have $H \supset (t(x)$ r $H)$ for some term t.* □

We can internally formalize this theorem in RPT. In the following, we say p is a Harrop propositional function if, for any term a, $p(a)$ is a Harrop proposition.

Definition 7 (Harrop propositional function)

$$\begin{array}{rcl} \texttt{HPF}_i & \triangleq & \lambda f.\lambda p.\ p = \lambda x.\ \texttt{true} \\ & & \vee\ p = \lambda x.\ \texttt{false} \\ & & \vee\ \exists y.\exists z.\ (p = (\lambda x.\ y(x) = z(x))) \\ & & \vee\ \exists q.\exists r.\ (p = (\lambda x.\ q(x) \wedge r(x)) \wedge f[q] \wedge f[r]) \\ & & \vee\ \exists q.\exists r.\ (p = (\lambda x.\ q(x) \supset r(x)) \wedge \forall x.\ (\models_i q(x)) \wedge f[r]) \\ & & \vee\ \exists q.\ (p = \lambda x.\ \forall y.\ q(\langle x, y\rangle) \wedge f[q]) \end{array}$$

Theorem 6 *We can prove the following for some term a.*

$$a \vdash_{i+2} \forall p.(\texttt{HPF}_i[p] \supset \forall x.(\models_{i+1} p(x)))$$

Theorem 7 *We can prove the following for some term a.*

$$a \vdash_{i+2} \forall p.(\texttt{HPF}_i[p] \supset \exists f.\forall x.(p(x) \supset (f(x) \vdash_{i+1} p(x))))$$

These two theorems are proved by the induction on the predicate $\texttt{HPF}_i$. The latter theorem is a formalized version of Theorem 5.

Theorems 6 and 7 still hold if we add clauses $\lambda x.\ a(x) \downarrow$ and $\lambda x.\ a(x)\ \&\ b(x)$ in the definition of $\texttt{HPF}_i$.

2.4 Strong Normalizability of RPT

In this section we study several metamathematical properties of the formal system RPT and its subsystem RPT_0. RPT_0 is essentially RPT without inductive definitions, so it can be said as the logical core of RPT.

2.4.1 Strong Normalizability and Weak Normalizability

Among many properties, the strong normalization (SN) property, every sequence of normalization is finite, is one of the most interesting one. Here, "normalization" means normalization of proof-terms or that of proof-figures. A normalization step of proof-terms is the same notion as reduction of terms. A normalization step of proof-figures in a natural deduction style logic is to eliminate a redundant part in proof-figures. For example,

$$\dfrac{\dfrac{\begin{array}{c}\Pi \\ a \vdash_i A\end{array} \quad \begin{array}{c}\Phi \\ b \vdash_i B\end{array}}{\langle a, b\rangle \vdash_i A \wedge B}\ \wedge\text{-intro}}{\texttt{car}(\langle a, b\rangle) \vdash_i A}\ \wedge\text{-elim}$$

This proof-figure is normalized to the following proof-figure.

$$\begin{array}{c}\Pi \\ a \vdash_i A\end{array}$$

With this process, the one-step reduction of proof-terms is associated:

$$\texttt{car}(\langle a, b\rangle) \to a$$

Therefore, normalization of proof-figures and that of proof-terms are closely connected. In most type theories, these two notions are identical. In RPT, the two notions are not identical. We will come back to this point later.

The normalization process is not deterministic; we can normalize any redundant part in a proof-figure, or any redex in a proof-term. The SN property is that any normalization process terminates. On the other hand, the WN (weak normalization) property is that, for any proof-figure (or proof-term), there exists a terminating normalization process. Obviously, the SN property subsumes the WN property.

The SN property holds for many logical systems and type theories, and is considered as one of the most important proof-theoretic properties.

2.4.2 Strong Normalizability of $\texttt{RPT}$

As shown in Theorem 2, even the weak normalization property fails for RPT.

The reason of this failure is similar to that in Martin-Löf's type theory; we can deduce anything from the falsity, therefore, assuming $\perp$ (the falsity), we can construct any proof figures which may not terminate.

Svensson observed that, if the normalization process is restricted so that the inside of λ-terms (of proof-terms) may not be reduced, then the SN property holds[41]. We will restrict the reduction in the same manner as she did.

2.4.3 Correspondence between Proof-figures and Proof-terms

In most type theories, a proof-figure and a proof-term 1-to-1 correspond to each other. In this case, normalization of proof-figures and that of proof-terms are the same process.

In the case of RPT, they are related, but do not have 1-to-1 correspondence for the reasons explained below. As we will describe in the following, the strong normalizability for the proof-terms does not hold while we can still prove the strong normalizability for the proof-figures in a subsystem of RPT. In order to obtain proof-theoretic properties such as consistency, it is sufficient to have the strong normalizability for the proof-figures. However, we will recover the 1-to-1 correspondence of the proof-terms and the proof-figures by introducing some auxiliary function symbols and modifying several inference rules of RPT. By recovering this correspondence, the strong normalizability of the proof-terms and that of the proof-figures are equivalent. We will henceforth prove the strong normalizability of the proof-terms of RPT only. However, our proof can be readily applicable to the strong normalizability of the proof-figures of the original RPT (without introducing new function symbols and modifying inference rules).

In this subsection, we shall analyze the three reasons why the 1-to-1 correspondence was lost in RPT, and show how to recover it.

The first reason is the existence of the following equal-left rule:

$$\frac{a_x[d] \vdash_i b \quad c \vdash_i d = e}{a_x[e] \vdash_i b}$$

By this rule, we can replace a proof-term to an equal term. It follows that, even if $a_x[d]$ is strongly normalizing, the resulting proof-term $a_x[e]$ is not guaranteed so. Hence, this rule is one source which destroys the correspondence. However, the application of the equal-left rule can be postponed as in the following lemma.

Lemma 1 *Given a proof-figure which consists of an application of the equal-left rule followed by an application of some rule, then we can transform it to a proof-figure which consists of applications of the latter rule and the equal-left rule. Namely, we can exchange the order of the application of inference rules.*

This lemma is easily proved by the case-analysis. By this lemma, it is meaningful to consider a system which lacks the equal-left rule. If we can prove SN of such a system, then we immediately have WN of the system with the equal-left rule.

The second reason is that, the level-up and level-down inferences do not introduce any function symbols so that the successive application of the level-up and level-down inferences is a redex in a proof-figure, but not a redex in a proof-term.

$$\frac{a \vdash_j p}{a \vdash_i (a \vdash_j p)}\ (i > j) \qquad \frac{a \vdash_i (b \vdash_j p)}{a \vdash_j p}\ (i > j)$$

A solution of this problem is to introduce a function symbol for each inference rule and a corresponding reduction rule:

$$\frac{a \vdash_j p}{\texttt{up}(a) \vdash_i (a \vdash_j p)}\ (i > j) \qquad \frac{a \vdash_i (b \vdash_j p)}{\texttt{down}(a) \vdash_j p}\ (i > j)$$

The corresponding reduction rule is $\texttt{down}(\texttt{up}(a)) \to a$.

The third reason is that some quantifier rules introduce terms in the righthand side of the provability sign into the lefthand side. For instance, recall that Ω is the term $(\lambda x.xx)(\lambda x.xx)$, and consider the following proof:

$$\frac{0 \vdash_0 \Omega = \Omega}{(\Omega, 0) \vdash_0 \exists x.x = x}$$

Since Ω does not have a normal form, we do not have the weak normalization. However, a redex in the term Ω do not correspond to a redex in the proof-figure (there is no redex in the proof-figure above), we do not have to consider the normalization process inside the term Ω. We will introduce a new function symbol `freeze` to make such a term freeze, that is, not reduced in a normalization step.

If we modify RPT as above, the lost 1-to-1 correspondence of proof-terms and proof-figures is recovered. Even if we do not have the 1-to-1 correspondence of the two, we can prove the strong normalization theorem of the proof-figures. However, by recovering the correspondence, our proof becomes much simpler than otherwise.

2.4.4 Formalizing RPT$_0$

We present RPT$_0$ as a variant of RPT. The motivation of this modification was described in the last subsection.

We first add three terms into those of RPT.

Definition 8 (New Term)

$$t \ ::= \ \texttt{up}(t) \mid \texttt{down}(t) \mid \texttt{freeze}(t)$$

Definition 9 (Neutral Term) *A term a is called neutral if it is of the form* $\texttt{car}(b)$, $\texttt{cdr}(b)$, $b(c)$, *or* $\texttt{down}(b)$, *and is not normal.*

Note that, a neutral term cannot be normal unlike the usual definition. This modification is crucial in our proof of the SN property.

Definition 10 (New Reduction Rules) *New reduction rules in RPT$_0$ are the following two rules:*

$$\begin{aligned} \texttt{down}(\texttt{up}(a)) &\to a \\ \texttt{freeze}(a) &\to a \end{aligned}$$

Note that, the relations $\to$, $\to^*$, and $\approx$ in this Section are those extended by these rules.

Theorem 1 *The new calculus satisfies the Church-Rosser property.*

Definition 11 (Restricted Reduction) *$a \twoheadrightarrow b$ if b is obtained by a sequence of reductions of the term a where no redex in $\lambda x.t$ nor $\texttt{freeze}(t)$ is reduced.*

Note that $\twoheadrightarrow$ is not Church-Rosser, since

$$(\lambda x.\lambda y.x)(\texttt{car}(\langle 0,0\rangle)) \twoheadrightarrow \lambda y.\texttt{car}(\langle 0,0\rangle),$$

and

$$(\lambda x.\lambda y.x)(\texttt{car}(\langle 0,0\rangle)) \twoheadrightarrow (\lambda x.\lambda y.x)(0) \twoheadrightarrow \lambda y.0.$$

The inference rules of RPT$_0$ are those of RPT with the the inductive definitions deleted, the following rule added, and the level-up, the level-down, the $\forall$-elim-1, and the $\exists$-intro rules modified.

$$\boxed{\dfrac{p \vdash_i \texttt{up}(a) = \texttt{up}(b)}{p \vdash_i a = b}}$$

The modification of the four rules are as follows:

$$\boxed{\begin{array}{cc} \dfrac{a \vdash_j p}{\texttt{up}(a) \vdash_i (a \vdash_j p)}\,(i>j) & \dfrac{a \vdash_i (b \vdash_j p)}{\texttt{down}(a) \vdash_j p}\,(i>j) \\[2ex] \dfrac{a \vdash_i \forall p}{a(\texttt{freeze}(t)) \vdash_i p(t)} & \dfrac{a \vdash_i p(t)}{\langle \texttt{freeze}(t), a\rangle \vdash_i \exists p} \end{array}}$$

The modification for these four rules are mainly introducing the function symbols `up`, `down`, and `freeze`.

The system RPT$_0^-$ is RPT$_0$ without the left-equal rule.

In the following, we shall prove the strong normalizability for RPT$_0^-$, and then obtain some proof-theoretic results for RPT$_0^-$ and $\mathcal{RPT}0$.

2.4.5 Reducibility and its properties

This subsection presents the main theorem of this section, from which we have the SN property for RPT$_0^-$.

The technique is based on Tait-Girard's method of computability predicates [19]. However, we are unable to directly apply the technique to RPT$_0^-$, since we cannot use the induction on the logical complexity of propositions. Instead, we will use the induction on the metaness level and the complexity of propositions.

Definition 12 (Reducibility Set) *For an i-th level proposition A, we define a reducibility set $Red_i(A)$ as a set of closed terms. This definition is by induction on the proof of $\models_i (A)$*

- *A is* `true`,

  $Red_i(A)$ is defined, and is equal to S where S is the set of closed terms which strongly normalize to 0,

- *A is* `false`,

  $Red_i(A)$ is defined, and is equal to $\{\}$.

- *A is $b = c$,*

  $Red_i(A)$ is defined, and is equal to S if $b \approx c$, and is equal to $\{\}$ otherwise.

- *A is $B \wedge C$,*

  $Red_i(A)$ is defined if and only if both $Red_i(B)$ and $Red_i(C)$ are defined. $a \in Red_i(A)$ holds if and only if there exist c and d such that $a \approx \langle c,d\rangle$, and $\texttt{car}(a) \in Red_i(B)$ and $\texttt{cdr}(a) \in Red_i(C)$.

- *A is $B \vee C$,*

  $Red_i(A)$ is defined if and only if both $Red_i(B)$ and $Red_i(C)$ are defined.

  $a \in Red_i(A)$ holds if and only if there exist c and d such that $a \approx \langle c,d\rangle$, and either $\texttt{car}(a) \approx$ `true` and $\texttt{cdr}(a) \in Red_i(B)$, or $\texttt{car}(a) \approx$ `false` and $\texttt{cdr}(a) \in Red_i(C)$.

- *A is $B \supset C$,*

  $Red_i(A)$ is defined if and only if $Red_i(B)$ is defined, and either $Red_i(B)$ is empty, or $Red_i(C)$ is defined.

  $a \in Red_i(A)$ holds if and only if there exist y and d such that $a \approx \lambda y.d$, a is strongly normalizing, and for all $b \in Red_i(B)$, $a(b) \in Red_i(C)$ holds.

- *A is $B \,\&\, C$,*

  $Red_i(A)$ is defined if and only if $Red_i(B)$ is defined, and either $Red_i(B)$ is empty, or $Red_i(C)$ is defined.

  $a \in Red_i(A)$ holds if and only if there exist c and d such that $a \approx \langle c,d\rangle$, and $\texttt{car}(a) \in Red_i(B)$ and $\texttt{cdr}(a) \in Red_i(C)$

- *A is $\forall B$,*

  $Red_i(A)$ is defined if and only if for every closed term b, $Red_i(B(b))$ is defined.

  $a \in Red_i(A)$ holds if and only if there exist y and d such that $a \approx \lambda y.d$, and for every closed term b, $a(\texttt{freeze}(b)) \in Red_i(B(b))$.

- *A is $\exists B$,*

  $Red_i(A)$ is defined if and only if for every closed term b, $Red_i(B(b))$ is defined.

  $a \in Red_i(A)$ holds if and only if there exist c and d such that $a \approx \langle c, d\rangle$, and $\texttt{cdr}(a) \in Red_i(B(\texttt{car}(a)))$.

- *A is $\models_j B$,*

  $Red_i(A)$ is defined if and only if $j < i$.

  $Red_i(A)$ is equal to S if $Red_j(B)$ is defined, and $\{\}$ otherwise.

- *A is $b \vdash_j B$,*

  $Red_i(A)$ is defined if and only if $j < i$.

  $a \in Red_i(A)$ holds if and only if $Red_j(B)$ is defined, $\texttt{down}(a) \in Red_j(B)$, *and there exists a term c such that $b \approx c$, and $c \in Red_j(B)$.*

- *if $A \approx B$ and $Red_i(B)$ has been defined, then $Red_i(A)$ is equal to $Red_i(B)$.*

This finished the definition of $Red_i(A)$.

This inductive definition is well-defined, since the set of terms A for which $Red_i(A)$ has been defined is monotone at each level i.

We then define the CR properties as in the standard method.

Definition 13 (The CR properties) *(CR1) if $a \in Red_i(A)$ then a is strongly normalizing.*

(CR2) if $a \in Red_i(A)$, and $a \twoheadrightarrow_1 a'$, then $a' \in Red_i(A)$.

(CR3) if a is neutral, closed, and for every a' such that $a \twoheadrightarrow_1 a'$, $a' \in Red_i(A)$, then $a \in Red_i(A)$.

Here, $a \twoheadrightarrow_1 a'$ is the one-step reduction for $\twoheadrightarrow$. We refer these properties as (CR).

Lemma 2 *For each natural number i and a proposition A, the set $Red_i(A)$ satisfies the CR properties.*

Proof.

The lemma is proved by the double induction; namely, the induction on the level, and the induction on the definition of the set $Red_i(A)$.

We assume that the lemma has been proved for each level $j < i$.

In the following, we do not mention the "well-typed" requirements in the definition of Red_i. For example, $a \in Red_i(B \wedge C)$ must be in the form of $\langle c, d\rangle$ for some c and d. In proving (CR2) and (CR3), it is automatically guaranteed by virtue of the modified definition of neutrality.

- A is $\texttt{true}$,

  Trivial.

- A is $\texttt{false}$,

  There is no $a \in Red_i(A)$, so CR trivially holds.

- A is $b = c$,

  If $b \approx c$, then $a \in Red_i(A)$ means a is strongly normalizing to 0, so CR holds. Otherwise, $Red_i(A)$ is empty, so CR holds.

- A is $B \wedge C$,

  1. Suppose $a \in Red_i(B \wedge C)$. We have $\texttt{car}(a) \in Red_i(B)$, and by the induction hypothesis, $\texttt{car}(a)$ is strongly normalizing, so is a.

  2. Suppose $a \in Red_i(B \wedge C)$, and $a \twoheadrightarrow_1 a'$. Since $\texttt{car}(a) \in Red_i(B)$ and $\texttt{car}(a) \twoheadrightarrow_1 \texttt{car}(a')$, $\texttt{car}(a') \in Red_i(B)$. Similarly, $\texttt{cdr}(a') \in Red_i(C)$, and then $a' \in Red_i(B \wedge C)$.

  3. Suppose a is neutral, and for every a' such that $a \twoheadrightarrow_1 a'$, $a' \in Red_i(B \wedge C)$. Since a is neutral, every 1-step-reduct of $\texttt{car}(a)$ is of the form $\texttt{car}(a')$ where $a \twoheadrightarrow_1 a'$. Since $\texttt{car}(a') \in Red_i(B)$, we have $\texttt{car}(a) \in Red_i(B)$ by the induction hypothesis. Similarly we have $\texttt{cdr}(b) \in Red_i(C)$ and then $a \in Red_i(B \wedge C)$.

- A is $B \vee C$,

  1. Suppose $a \in Red_i(B \vee C)$. We have $\texttt{cdr}(a) \in Red_i(B)$, or $\texttt{cdr}(a) \in Red_i(C)$. By the induction hypothesis, $\texttt{cdr}(a)$ is strongly normalizing in either case, so is a.

  2. Suppose $a \in Red_i(B \vee C)$, and $a \twoheadrightarrow_1 a'$. Assume $\texttt{car}(a) \approx \texttt{true}$. Since $\texttt{cdr}(a) \in Red_i(B)$ and $\texttt{cdr}(a) \twoheadrightarrow_1 \texttt{cdr}(a')$, we have $\texttt{cdr}(a') \in Red_i(B)$. We also have $\texttt{car}(a') \approx \texttt{true}$, so we have $a' \in Red_i(B \vee C)$. Similarly for the case of $\texttt{car}(a) \approx \texttt{true}$.

  3. Suppose a is neutral, and for every a' such that $a \twoheadrightarrow_1 a'$, $a' \in Red_i(B \vee C)$. Fix such an a'. Then, either $\texttt{car}(a') \approx \texttt{true}$ and $\texttt{cdr}(a') \in Red_i(B)$, or $\texttt{car}(a') \approx \texttt{false}$ and $\texttt{cdr}(a') \in Red_i(C)$. Assume the former is the case. Since a is neutral, every 1-step-reduct of $\texttt{cdr}(a)$ is of the form $\texttt{cdr}(a')$ where $a \twoheadrightarrow_1 a'$. Since $\texttt{cdr}(a') \in Red_i(B)$, we have $\texttt{cdr}(a) \in Red_i(B)$ by the induction hypothesis. We also have $\texttt{car}(a) \approx \texttt{true}$, so $a \in Red_i(B \vee C)$.

  Similarly for the latter case.

- A is $B \supset C$,

  1. Trivial.

  2. Suppose $a \in Red_i(B \supset C)$, and $a \twoheadrightarrow_1 a'$. For every $b \in Red_i(B)$, $a(b) \in Red_i(C)$ holds. By the induction hypothesis, $a'(b) \in Red_i(C)$ holds for all such b. Also a' is clearly strongly normalizing, so $a' \in Red_i(B \supset C)$.

3. Suppose a is neutral, and for every a' such that $a \twoheadrightarrow_1 a'$, we have $a' \in Red_i(B \supset C)$. Fix such an a'. Since a' is strongly normalizing, so is a. For every $b \in Red_i(B)$, we have $a'(b) \in Red_i(C)$. Since a is neutral, $a(b)$ reduces to a term of the form $a'(b)$ or $a(b')$ where $b \twoheadrightarrow_1 b'$. By the induction hypothesis, b is strongly normalizing, so we eventually get $a'(b'') \in Red_i(C)$ by the induction on the length of the normalizing sequence of b. Finally, we have $a \in Red_i(B \supset C)$.

- A is $B \,\&\, C$,

  Similar to the case of $\wedge$.

- A is $\forall B$,

  1. Suppose $a \in Red_i(\forall B)$. Then, we have $a(\texttt{freeze}(0)) \in Red_i(B(0))$, hence $a(\texttt{freeze}(0))$ is strongly normalizing, so is a.

  2. Suppose $a \in Red_i(\forall B)$, and $a \twoheadrightarrow_1 a'$. Since $a(\texttt{freeze}(b)) \twoheadrightarrow_1 a'(\texttt{freeze}(b))$, $a'(\texttt{freeze}(b)) \in Red_i(B(b))$ for every closed term b. So $a' \in Red_i(\forall B)$.

  3. Suppose a is neutral, and for every a' such that $a \twoheadrightarrow_1 a'$, we have $a' \in Red_i(\forall B)$. So, $a'(\texttt{freeze}(b)) \in Red_i(B(b))$ for any closed term b. Since a is neutral, $a(\texttt{freeze}(b))$ reduces to the form $a'(\texttt{freeze}(b))$, so $a(\texttt{freeze}(b)) \in Red_i(B(b))$. Hence, we have $a \in Red_i(\forall B)$.

- A is $\exists B$,

  1. Suppose $a \in Red_i(\exists B)$. We have $\texttt{cdr}(a) \in Red_i(B(\texttt{car}(a)))$, and by the induction hypothesis, $\texttt{cdr}(a)$ is strongly normalizing, so is a.

  2. Suppose $a \in Red_i(\exists B)$, and $a \twoheadrightarrow_1 a'$. Since $\texttt{cdr}(a) \twoheadrightarrow_1 \texttt{cdr}(a')$, $\texttt{cdr}(a') \in Red_i(B(\texttt{car}(a)))$. So we have $\texttt{cdr}(a') \in Red_i(B(\texttt{car}(a')))$, hence $a' \in Red_i(\exists B)$.

  3. Suppose a is neutral, and for every a' such that $a \twoheadrightarrow_1 a'$, we have $a' \in Red_i(\exists B)$. Since a is neutral, every 1-step-reduct of $\texttt{cdr}(a)$ is of the form $\texttt{cdr}(a')$ where $a \twoheadrightarrow_1 a'$. Since $\texttt{cdr}(a') \in Red_i(B(\texttt{car}(a')))$, we have $\texttt{cdr}(a) \in Red_i(B(\texttt{car}(a)))$. Then we have $a \in Red_i(\exists B)$.

- A is $\models_j B$,

  Similarly to the case of $a = b$.

- A is $b \vdash_j B$,

  1. Suppose $a \in Red_i(b \vdash_j B)$. Then, $\texttt{down}(a) \in Red_j(B)$, so by the induction hypothesis, a is strongly normalizing.

  2. Suppose $a \in Red_i(b \vdash_j B)$, and $a \twoheadrightarrow_1 a'$. Then $\texttt{down}(a) \in Red_j(B)$, and by the induction hypothesis, $\texttt{down}(a') \in Red_j(B)$. Hence, $a' \in Red_i(b \vdash_j B)$.

  3. Suppose a is neutral, and for every a' such that $a \twoheadrightarrow_1 a'$, $a' \in Red_i(b \vdash_j B)$. Then, $\texttt{down}(a') \in Red_j(B)$. Since a is neutral, every 1-step-reduct of $\texttt{down}(a)$ is of the form $\texttt{down}(a')$, and by the induction hypothesis, $\texttt{down}(a) \in Red_j(B)$. Hence, we have $a \in Red_i(b \vdash_j B)$.

- if $A \approx B$ and $Red_i(B)$

  Since we have the Church-Rosser Theorem for $\approx$, this case is trivial.

This completes the proof. □

We define $Trans(J)$ for a judgement J as follows:

- $Trans(\models_i a)$ is "$Red_i(a)$ is defined", and
- $Trans(a \vdash_i b)$ is "$Red_i(b)$ is defined and $a \in Red_i(b)$".

The following theorem is the main theorem of this chapter.

Theorem 2 *If we have a proof of the judgement J with assumptions $J_1, \cdots, J_n$ in RPT_0^-, then, we have that $Trans(J_1\theta), \cdots, Trans(J_n\theta)$ imply $Trans(J\theta)$ where θ is any ground substitution.*

In this theorem, a ground substitution is a substitution where substituted terms are closed terms only.

Proof.

This theorem is proved by the induction on the length of the proof.

For a strongly normalizing term a, we will use the notation $len(a)$ which represents the maximum of length of reduction sequences starting from a.

The induction proceeds by the case analysis of the last inference rule. In the following, we omit the trivial cases and list non-trivial cases only. In general, inference rules for levels, equality and canonical terms, and propositions are easily handles, since they do not contain proof-terms. Moreover, most elimination inference rules (such as the $\wedge$-elim-1 rule) are straightforward, since the definition of $Red_i(A)$ is in the form of the elimination style.

Case (prop-intro).

Since $Red_j(p)$ is defined, and $0 \in S$, we have the conclusion.

Case (level-up).

By the reduction $\texttt{down}(\texttt{up}(a)) \to a$, we have the conclusion.

Case (level-down).

This rule is essentially an elimination rule, so it is straightforward.

Case ($\wedge$-intro).

Suppose $a \in Red_i(p)$, and $b \in Red_i(q)$ for closed terms a, b, p, q and an ordinal number i which is less than α.

Consider the term $\texttt{car}(\langle a, b\rangle)$. a and b are strongly normalizing. By the induction on $len(a) + len(b)$, we have $\langle a, b\rangle \in Red_i(p \wedge q)$.

(Basis) a and b are normal. Then the only 1-step reduction of $\mathtt{car}(\langle a, b\rangle)$ is a, which is in $Red_i(p)$.

(Step) $\mathtt{car}(\langle a, b\rangle)$ 1-step reduces to $\mathtt{car}(\langle a', b\rangle)$, $\mathtt{car}(\langle a, b'\rangle)$, or a. As for the first two cases, since $a' \in Red_i(p)$ and $b' \in Red_i(q)$ hold, we have the conclusion using the induction hypothesis. As for the third case, we have $a \in Red_i(p)$. Therefore, in any case, the results of 1-step reduction are all in $Red_i(p)$, so is $\mathtt{car}(\langle a, b\rangle)$.

Similarly, we can prove $\mathtt{cdr}(\langle a, b\rangle) \in Red_i(q)$, and hence, $\langle a, b\rangle \in Red_i(p \wedge q)$.

Case ($\vee$-intro-1).

Suppose $a \in Red_i(p)$ and $Red_i(q)$ is defined. Then, we have $Red_i(p \vee q)$ is defined.

The term $\mathtt{cdr}(\langle\mathtt{true}, a\rangle)$ 1-step reduces to $\mathtt{cdr}(\langle\mathtt{true}, a'\rangle)$ or a. For the first case, we can use induction hypothesis to prove $\mathtt{cdr}(\langle\mathtt{true}, a'\rangle) \in Red_i(p)$ since a is strongly normalizing. For the latter case, we already have $a \in Red_i(p)$. Hence we have $\mathtt{cdr}(\langle\mathtt{true}, a\rangle) \in Red_i(p)$ and get the conclusion.

Case ($\vee$-intro-2).

Similarly to the Case $\vee$-intro-1.

Case ($\vee$-elim).

Suppose $a\theta \in Red_i(p\theta \vee q\theta)$. Suppose further, if $0 \in Red_i(\mathtt{car}(a\phi) = \mathtt{true})$ and $x\phi \in Red_i(p\phi)$, then $b\phi \in Red_i(r\phi)$, and if $0 \in Red_i(\mathtt{car}(a\psi) = \mathtt{false})$ and $x\psi \in Red_i(q\psi)$, then $c\psi \in Red_i(r\psi)$ for any ground ϕ and ψ.

From the first assumption, we have either $\mathtt{car}(a\theta) \approx \mathtt{true}$ and $\mathtt{cdr}(a\theta) \in Red_i(p\theta)$, or $\mathtt{car}(a\theta) \approx \mathtt{false}$ and $\mathtt{cdr}(a\theta) \in Red_i(q\theta)$. Suppose we have the first case. Using the second assumption with ϕ being θ with $x := \mathtt{cdr}(a\theta)$, we get $b_x[\mathtt{cdr}(a)]\theta \in Red_i(r\theta)$ with any ground θ.

We will prove $(\mathit{if}\ \mathtt{car}(a)\ \mathit{then}\ \lambda x.b\ \mathit{else}\ \lambda x.c)(\mathtt{cdr}(a))\sigma \in Red_i(r\sigma)$ for any ground σ by the induction on $len(\mathtt{car}(a)\sigma) + len(\mathtt{cdr}(a)\sigma)$.

(Base) $\mathtt{car}(a)\sigma$ must be $\mathtt{true}$, so the whole term is reduced to $(\lambda x.b)(\mathtt{cdr}(a))\sigma$, and then $b_x[\mathtt{cdr}(a)]\sigma$. Then, by the fact above, this is in $Red_i(r\sigma)$.

(Step) The whole term can be reduced at redexes $\mathtt{car}(a)\sigma$, the $\mathtt{if}$ term, or $\mathtt{cdr}(a)\sigma$. In the first and the third cases, we can use the induction hypothesis. In the second case, the term becomes $(\lambda x.b)(\mathtt{cdr}(a))\sigma$. We can prove this term is in $Red_i(r\sigma)$ by the induction on $len(\mathtt{cdr}(a)\sigma)$.

Similarly for the other case.

This finishes the Case $\vee$-elim.

Case ($\supset$-intro).

Suppose $Red_i(p)$ is defined, and for any ground substitution θ, if $x\theta \in Red_i(p\theta)$, then $a\theta \in Red_i(q\theta)$. Under the restricted reduction, we have $(\lambda x.a)\theta$ is normal. Then, all we have to prove is that, for any $b \in Red_i(p\theta)$, we have $((\lambda x.a)\theta)(b) \in Red_i(q\theta)$. We can prove this by induction of $len(a\theta) + len(b)$.

Case (&-intro).

Similar to the case of $\wedge$.

Case ($\forall$-intro).

Suppose $a\theta \in Red_i(p\theta)$ for any ground θ.

We will show $(\lambda x.a)(\mathtt{freeze}(b))\phi \in Red_i((\lambda x.p)b\phi)$ for any ground ϕ. The key case is $a_x[\mathtt{freeze}(b)]\phi$. By taking θ as ϕ with $x := \mathtt{freeze}(b)$, we have the result.

Case ($\exists$-intro). Suppose $a\theta \in Red_i(p_x[t]\theta)$ for any ground θ.

We will show $\mathtt{cdr}(\langle\mathtt{freeze}(t), a\rangle)\phi \in Red_i((\lambda x.p)\mathtt{car}(\langle\mathtt{freeze}(t), a\rangle)\phi)$ for any ground ϕ by the induction on $len(a)$. Note that $Red_i((\lambda x.p)\mathtt{car}(\langle\mathtt{freeze}(t), a\rangle)\phi)$ is equal to $Red_i(p_x[t])\phi$.

The term $\mathtt{cdr}(\langle\mathtt{freeze}(t), a\rangle)\phi$ can be reduced to either $a\phi$ or $\mathtt{cdr}(\langle\mathtt{freeze}(t), a'\rangle)\phi$ where $a\phi \twoheadrightarrow_1 a'\phi$. In the first case, we have the result by setting θ in the assumption be ϕ. In the second case, we also have the conclusion by using the induction hypothesis.

This finishes the proof. □

2.4.6 Properties of $\mathbf{RPT}_0^-$

Theorem 3 *If $a \vdash_i b$ is proved in RPT_0^- without assumptions, a is strongly normalizing in the sense of $\twoheadrightarrow$, or equivalently, a proof figure of $a \vdash_i b$ is strongly normalizing.*

Proof.
From the last theorem, we have $a\theta \in Red_i(b\theta)$ for any ground substitution θ. Hence $a\theta$ is strongly normalizing (in the sense of $\twoheadrightarrow$) for any such θ. It follows that a is strongly normalizing. □

Theorem 4 *RPT_0^- is consistent.*

Proof.
By the theorem above, if we had a proof of $a \vdash_i \mathtt{false}$, then $Red_i(\mathtt{false})$ would have an element, but it contradicts to the definition of $Red_i(\mathtt{false})$.

Corollary 3 (Subformula property) *If $a \vdash_i b$ is proved in RPT_0^- without assumptions, all the propositions in its proof figure are sub-propositions of b.*

Our definition of Red_i contains "well-typed" statements so that we have the following result.

Theorem 5 *The proof terms are "well-typed" in the following sense;*

1. if $a \vdash_i b \wedge c$, $a \vdash_i b$ & c, or $a \vdash_i \exists b$ is proved in RPT_0^- without assumptions, then a is a pair, namely, is equal to a term of the form $\langle b, c\rangle$.

2. if $a \vdash_i b \vee c$, then a is a pair, and its car-part is equal to either $\mathtt{true}$ *or* $\mathtt{false}$.

3. if $a \vdash_i b \supset c$, or $a \vdash_i \forall x.b$ is proved in RPT_0^- without assumptions, then a is a function, namely, is equal to a term of the form $\lambda y.c$.

Corollary 4 (Disjunction Property) *If $a \vdash_i b \vee c$ is proved in RPT_0^- without assumptions, then we have either* $\mathtt{cdr}(a) \vdash_i b$ *or* $\mathtt{cdr}(a) \vdash_i c$ *in RPT_0^-.*

Proof.
From the main theorem, we have $a \in Red_i(b \vee c)$. By the definition of Red_i, a is reduced to the form of (d, e), and that $d \sim \texttt{true}$, or $d \sim \texttt{false}$. Since d is also strongly normalizing, we can reduce d as far as possible, and will get `true` or `false`. In the first case, we can make a proof of $e \vdash_i b$, and in the second case, $e \vdash_i c$.

2.4.7 Properties of $\mathbf{RPT_0}$ and RPT

By Lemma 1, a proof in $\mathrm{RPT_0}$ can be equivalently transformed into a proof which contains the equal-left rule at the last inference only. Hence, we have the following theorems.

Theorem 6 *The proof terms in RPT_0 is weakly normalizing.*

Theorem 7 *RPT_0 is consistent. The subformula property and the disjunction property hold for RPT_0.*

As we stated before, $\mathrm{RPT_0}$ is not exactly equivalent to RPT without inductive definition. But the structure of proof-figures of RPT is the same as that of $\mathrm{RPT_0}$, and we can prove the SN theorem for the proof-figure of RPT without inductive definitions and the equal-left rules in the same manner as $\mathrm{RPT_0^-}$.

Theorem 8 *RPT without the inductive definition is consistent. The subformula property and the disjunction property hold for RPT without the inductive definition.*

Remark 2 *In order to obtain proof-theoretic results about the full RPT, namely, RPT with arbitrary inductive definitions, we have to extend our results to include inductive definitions. It is our future work.*

2.5 Conclusion

In this chapter, we have presented RPT, a formal system of Sato's $\mathcal{RPT}$. By putting three reasonable conditions, RPT does correspond to $\mathcal{RPT}$.

There have been studied other constructive logics; Martin-Löf's type theory ITT_n[29], Feferman's T_0 [18], Hayashi and Nakano's PX [21], Sato's $\mathcal{SST}$[35], and Coquand and Huet's Calculus of Constructions (CoC)[11]. ITT_n and CoC have type theories while T_0, PX, and $\mathcal{SST}$ are untyped theories. Since RPT is based on untyped λ-calculus, we can define arbitrary partial recursive functions in RPT. Moreover, RPT gives an interpretation of logical connectives under the Curry-Howard isomorphism, so it has a feature of type theories. Therefore, wa may say RPT has features of both type and untyped theories.

The reflection mechanism of RPT is quite useful as was demonstrated by many examples in this Chapter. In particular, we formalized and proved many metatheorems such as Term Existence Property and Disjunction Property. We also formalized a metatheorem about Harrop formulas in the first-order logic, which is useful in program refinement techniques.

We also studied proof-theoretic properties of RPT without inductive definitions. To do so we presented a formal system $\mathrm{RPT_0}$ and $\mathrm{RPT_0^-}$. Every proof-figure of $\mathrm{RPT_0}$ is transformed into a proof-figure of $\mathrm{RPT_0^-}$ and an application of the equal-left rule. The calculus of $\mathrm{RPT_0^-}$ is so designed that the redexes in the proof term one-to-one correspond to the redexes in the proof-figure.

We defined a modified "reducibility candidate" property, and proved $Red_i(p)$ satisfies it. From this theorem, we got several fundamental proof-theoretic properties for $\mathrm{RPT_0^-}$, including the strong normalization property in a restricted calculus, the consistency, and the subformula property. We proved the similar properties for $\mathrm{RPT_0}$, too. The same technique can be applied to RPT itself, hence these results indicate that our formalization is a reasonable one. We have not proved proof-theoretic properties for the full RPT system (with inductive definitions). It is our future work.

Chapter 3

Constructive Programming System based on RPT

3.1 Introduction

In this chapter, we describe our Constructive Programming System for the formal system RPT.

A Constructive Programming System is a computer software which provides supports for men to develop a program (proof) in the paradigm of Constructive Programming. There are two reasons why we need a Constructive Programming System.

The first one is to ensure the correctness. In order to do Constructive Programming, we need to give a correct proof of a specification formula. If the proof contains errors, then the correctness of the extracted program is not guaranteed. Therefore we need a mechanical proof-checker which checks each inference step.

The second one is that a proof of a realistic specification tends to be quite large. Moreover it often contains many similar parts. Therefore it would be quite helpful for a computer software to provide some supports to men in developing a large proof.

We have designed and implemented a Constructive Programming System which supports proof-development in RPT. Our system also contains a certain level of automatic proof generation.

We will describe an overview of the Constructive Programming System. We also give a proof of Church-Rosser theorem for the terms in RPT, and finally give an example of Constructive Programming.

3.2 Overview of the System

The target logical system of our implementation is RPT. It follows that our target (object) programming language is Λ. We also chose Λ as the implementation language of our system, so the two languages are identical in our case.

Since Edinburgh LCF[20] was implemented on top of ML, almost all the proof development systems have been implemented by much stronger (much more expressive) programming languages than the object language which the system can reason about. In the case of LCF, ML is much stronger than the object language `PPlambda`. If we would want to prove some properties of the system itself, we would need to design an even stronger logical system. Even worse, this process does not terminate.

On the other hand, the object and implementation languages of our system are identical. We can therefore express properties of our system itself inside RPT, and then prove them using our system.

Our implementation language is slightly extended from the original Λ; we introduced the pattern-matching mechanism, assignment statements, and the error-handling mechanism. However the pattern-matching mechanism is just a syntax-sugar, and can be always eliminated from the program. The main role of the assignment statement is to keep the history of already proved results, therefore dereference of a variable (to which the value of a past proof was assigned) can be eliminated by the substitution of the variable by the past proof. Finally, the error-handling mechanism is not invoked if the proof-checker succeeds (namely our proof is correctly formulated). Hence, if we successfully prove some theorem using our system, then it can be regarded as an output of a system which is implemented by a pure language of Λ.

Each term in the object language must have a representation in the implementation language. In our case every term in Λ must have a representation in Λ itself. The representation must be a normal term, so the representation function cannot be the identify. We use the `quote` mechanism for the representation function. The (meta)function `quote` one-to-one-maps every term in Λ to pair-terms. The pair-terms are terms constructed by `nil` and $\langle \_, \_ \rangle$ only. Unlike the `quote` mechanism in Lisp, the result of the computation of `quote`(a) in Λ is `quote`(a) itself.

For improving the readability, our system uses the Japanese character-set for displaying logical connectives. However, we cannot input those characters without a Japanese-input method. Hence, the input by a user and the output from the system differ. We list this difference in Table 3.1.

As shown in the table above, if we input the judgement $s \vdash_i t$, then the proof-term s is not displayed, hence the output of our system looks like ordinary first-order logic. This is useful since we usually do not care the structure of the proof-term of the current theorem. In particular, we do not want to explicitly name an assumption variable when we assume some proposition. Instead, the system generates an appropriate name for the assumption, and its name is not shown in the display. However, the proof-term does exist inside the system. We can show it explicitly by giving a command to the system (the "proof" command).

RPT	Input to the system	Output from the system
$\perp$	`false`	$\perp$
T	`true`	T
$a = b$	`(eql` a b`)`	a `=` b
$p[a]$	`(pred` p a`)`	$p[a]$
$A \wedge B$	`(and` A B`)`	$A \wedge B$
$A \& B$	`(cand` A B`)`	A & B
$A \vee B$	`(or` A B`)`	$A \vee B$
$A \supset B$	`(imp` A B`)`	$A \supset B$
$\forall$x.A	`(all (lambda (`x`)` A`))`	$\forall$ x . A
$\exists$x.A	`(ex (lambda (`x`)` A`))`	$\exists$ x . A
$\models_i p$	`|=` p	`|=` p
$a \vdash_i p$	`|-` p	`|-` p

Table 3.1: Logical Connectives in Input and Output

3.2.1 Interaction with the system

Our system is an interactive proof generator, not a proof-checker (which checks a proof after inputing a complete proof), nor an automatic prover (which generates a proof from a formula automatically). The direction of our inference is forward. We construct a proof from leaves to the root.

Commands at the toplevel of the system are those corresponding to the inference rules of RPT as well as commands which shows the hidden information (the current level and the current proof-term).

```
RPT:100> true-intro
Result:100: |- T
RPT:101> proof
proof-term is 0
RPT:101>
```

In this example, `RPT:100>` is a prompt of our system. The number `100` is the history number, the number of proofs so far generated since ths system started. In reply to this prompt, the name of an inference rule, `true-intro`, was input by a user, then the system returned $0 \vdash_0$ `true`. The expression `Result:100:` is the header of this reply and the actual content is `|-` T only, which is the righthand side of the sequent $0 \vdash_0$ `true`. The proof-term 0 and the level 0 are not shown. To display the proof-term, the user inputs the command `proof`, and then got the result. Note that, this command did not increase the history number, since no new proof has been generated.

Commands which correspond to inference rules of RPT usually take several arguments, and return the righthand side of the inferred sequent if succeeds. The arguments specify the subproofs of this application, and other necessary information so that the system can uniquely identify the form of the inference. If the command does not succeed, namely, the application of the inference rule is not appropriate, then it does not return anything and raises an error.

For most arguments, default values are supplied by the system if no arguments are provided by the user. For instance, the following input means that, to apply the $\wedge$-introduction rule to the subproofs numbered 70 (two steps before this application) and 71 (one step before this application).

```
RPT:72> (and-intro -2 -1)
```

Since these are the default values of this rule, we can simply input as follows:

```
RPT:72> and-intro
```

If the system cannot supply default valued, it displays another prompt. For instance, the following example shows that the system is waiting for the user to input a term, since the `prop-eql` rule (showing that $a = b$ is a proposition) is applicable to any term, and the system does not know which term should be used at this point.

```
RPT:74> prop-eql
term?
```

Our system does not merely provide interactive proof-checker/proof-generator; it also supports facility of computation, and program extraction. Moreover, we extended the system RPT with some derived rules. For example, the $\exists$-elimination rule in the style of the usual first-order logic is not primitive, but a derived rule. Since it is a useful rule, we included it as a primitive command. Another example is the $\wedge$-introduction rule for more than 2 propositions. We can introduce a conjunctive proposition which consists of more than 2 conjuncts at a time. Similar derived rules are available for the elimination, and for other logical connectives such as $\vee$. Since these rules are not primitive inference rules in RPT, and are not guaranteed by any rigid way, we implemented these rules as combination of primitive inference rules. Obviously this implementation is inefficient, since it always expands the rule, and checks the expanded form while the validity of the derived rules are clear. This point is an important motivation of introducing more powerful reflection mechanism in Chapter 4.

A concrete example of our system will be given later.

3.2.2 Automatic Proof Generation

Although our system does not aim to generate proofs automaticly, several simple automatic proving procedures have been built-in our system.

- Proving some term is a proposition

  A characteristic point of RPT is that, the propositionhood is not defined solely, but it depends on the truthhood. For instance, $a \supset 0$ is a proposition if a is a false proposition. Hence, we need a proof for propositionhood.

  However, we can define a large class of terms where the propositionhood does not depend on the truthhood, hence is decidable. Namely, for terms $a \supset b$ and $a \,\&\, b$, if both subterms a and b are propositions, so are the whole terms. Our system contains this decision algorithm. Therefore, in most cases, the propositionhood is automatically proved.

- Computation

  Given a representation (`quote`'d form) of a term a, the system can compute it at any time. This computation is achieved by the formalized interpreter and does not depend on the interpreter of Λ itself. Hence, we can arbitrarily change the computation strategy by giving some message to the system.

- Simplification of propositions

  Our simplifier operates on propositions, and transforms them to logically equivalent ones. It consists of normal simplification and E-simplification.

  Normal simplification is to transform a proposition to a simpler form. For instance, $\texttt{true} \wedge A$ can be simplified to A, and $\texttt{car}(\langle A, B\rangle)$ to A.

  We then describe E-simplification.

  We call a proposition of the form $\exists x_1, \cdots, \exists x_n.(A_1 \vee \cdots \vee A_m)$ an E-proposition. Suppose some A_i is an equality proposition $a = b$. For instance, the following is an E-proposition:

  $$e_1 \triangleq \exists x.\exists y.\exists z.(u = \langle x, y\rangle \wedge y = \langle 0, z\rangle \wedge P[z])$$

  Let p be a proposition, and e be an E-proposition. Suppose $a \vdash_i p$ and $b \vdash_i e$ have already been proved. If the following procedure succeeds with a new proposition r, then we obtain a proof of $a \vdash_i r$ by E-simplification.

  1. Rename all the bound variables in e so that they do not crash with free variables in p and e.
  2. Let S be the set of equalities in (the atomic formulas of) e. Let θ be the most general unifier of S.
  3. Apply the normal simplifier to $p\theta$. Let r be the resulting proposition.

4. If no bound variable in e appears in r, then the procedure succeeds with r. Otherwise, it fails.

As a concrete example of the E-simplification, Let e_1 be as above, and p_1 be $\texttt{pair?}(u) = \texttt{true} \wedge \texttt{fun?}(\texttt{cdr}(u)) = \texttt{true} \vee p_2$.

We assume that p_2 does not contain u free. The most general unifier of the set of equalities in e_1 is

$$\theta_1 \triangleq \{u := \langle x, \langle 0, z\rangle\rangle, y := \langle 0, z\rangle\}$$

Then $p_1\theta$ is $\texttt{pair?}(\langle x, \langle 0, z\rangle\rangle) = \texttt{true} \wedge \texttt{fun?}(\texttt{cdr}(\langle x, \langle 0, z\rangle\rangle)) = \texttt{true} \vee p_2$, and then it simplifies to p_2. Since, p_2 does not contain variables x, y and z, the result of E-simplification of p_1 is p_2.

E-simplification does not do much work. However, E-propositions are often contained in inductively defined predicates, so we can make use of E-simplification at the proof of induction steps. In particular, we used it extensively in proving the Church-Rosser theorem of Λ.

3.3 Mechanized Proof of the Church-Rosser Theorem

The Church-Rosser Property is one of the most fundamental properties for term rewriting systems and functional programming languages. Let D be a set and R be a binary relation on D. Let $=_R$ be an equivalence relation induced by R. Then R is Church-Rosser if, for any terms $a, b, c \in D$ such that $a =_R b$ and $a =_R c$ hold, there exists a term $d \in D$ such that bRd and cRd hold.

This property guarantees that the result of any computation from a term a is unique. The uniqueness of computation is significant for term rewriting systems and functional languages to have a meaningful semantics.

A similar property is the following Diamond property.

Definition 14 (Diamond Property) *Let D be a set and R be a binary relation on D.*

Then R has the diamond property if for any terms $a, b, c \in D$ such that aRb and aRc hold, there exists a term $d \in D$ such that bRd and cRd hold. □

If R satisfies the Diamond property, then it satisfies the Church-Rosser property. Hence, we will concentrate on the diamond property in the following.

There are several works in which mechanical proofs of the Church-Rosser property have been given. For example, Shankar[38] proved the Church-Rosser property for the pure λ calculus using the famous Boyer-Moore theorem prover[9].

The difference of his work and ours is that, we prove the Church-Rosser property for our programming language Λ itself, and we use the novel technique due to Takahashi.

3.3.1 Proof Method

There have been proposed many techniques to prove the Church-Rosser property. Among them, Takahashi's method[42] is one of the best one as far as we know. Her method is quite simple, yet applicable for a wide range of reduction systems. Sato[34] used it to prove the Church-Rosser property of Λ. The method is summerized as follows:

1. Define a parallel reduction $\rightarrow$ of terms as an extension of the original reduction.

   Parallel reduction is such a reduction that reduces an arbitrary number of redexes at the same time. Since it does not specify the number and the positions of redexes, this reduction is non-deterministic. It is called "parallel", since it can reduce more than one redexes at a time. For instance, if we parallel-reduces the term $(\lambda x.xx)(\texttt{car}(\langle y, z\rangle))$, then the result is one of this term itself, $(\texttt{car}(\langle y, z\rangle))(\texttt{car}(\langle y, z\rangle))$, $(\lambda x.xx)y$, or yy.

   The parallel reduction must contain the original reduction, and must be contained in the reflexive-transitive closure of the original reduction.

2. For each term a, define the "most reduced" term a^*.

   Intuitively, the "most reduced" term is such a term that, all the redexes in a are reduced simultaneously. For instance,

$$((\lambda x.xx)(\texttt{car}(\langle y, z\rangle)))^* \equiv yy$$

3. Prove the following properties on $\rightarrow$ and *.

   - $a \rightarrow a$
   - $a \rightarrow c$ and $b \rightarrow d$ imply $a_x[b] \rightarrow c_x[d]$
   - $a \rightarrow b$ implies $b \rightarrow a^*$

   Then we have that $\rightarrow$ satisfies the Diamond property.

4. Show that, if a relation has the Diamond property, so does its reflexive and transitive closure.

We have mechanized Sato's proof in our system as follows:

1. Define the representation of the terms of Λ as pairs.

   All the terms in Λ are already represented as pairs by the `quote` mechanism. But we must again represent the terms of Λ in order to treat their properties.

   The `quote` mechanism is again used for this representation. Based on this representation, we define predicates for (representation of) terms, parallel reductions, and so on.

2. Prove that the parallel reduction has the Diamond property.

3. Prove that the reflexive-transitive closure of $\rightarrow$ has the Diamond property.

Obviously, the reflexive-transitive closure of $\rightarrow$ coincides the original reduction $\rightarrow^*$, therefore, this finishes the proof of the Church-Rosser property of $\rightarrow^*$.

The mechanized proof that $\rightarrow$ has the Diamond property is by induction on the structure of the term. Since Λ has various kinds of term construction, this induction needed many cases as the induction steps. However, many of them are similarly proved, and the essential complexity was not so high. We used the E-simplification procedure to prove each case of the induction steps.

In the following, we will describe the last part of the proof.

Theorem 8 *We can prove the following for some term a:*

$$a \vdash_1 \forall S.\forall A.(\mathtt{Unary}(S) \supset \mathtt{Binary}(A) \supset \mathtt{Persis}(A,S) \supset$$
$$\mathtt{Diamond}(A,S) \supset \mathtt{Diamond}(\mathtt{Trans}(A),S))$$

where

$$\begin{aligned}
\mathtt{Unary}(S) &\triangleq \forall x.\models_0 S[x] \\
\mathtt{Binary}(A) &\triangleq \forall x.\forall y.\models_0 A[x,y] \\
\mathtt{Trans}(A) &\triangleq \lambda f.\lambda x.\lambda y.(x = y \vee \exists z.A[x,z] \wedge f[z,y]) \\
\mathtt{Persis}(A,S) &\triangleq \lambda x,y.(A[x,y] \supset S[x] \supset S[y]) \\
\mathtt{Diamond}(A,S) &\triangleq \forall x.(S[x] \supset \forall y.\forall z.(A[x,y] \wedge A[x,z] \supset \exists u.(A[y,u] \wedge A[z,u])))
\end{aligned}$$

□

In this theorem, the unary predicate S represents the (quote'd) termhood, and Persis(A,S) means that the binary relation A respects the termhood S. This theorem means that, if a binary relation A has the Diamond property, then its transitive closure Trans(A) also has the Diamond property.

Theorem 8 holds for *any* binary relation, hence we can apply it to relations other than $\rightarrow$. This kind of generality is one of the characteristic points of RPT.

The proof of Theorem 8 has been mechanically checked by our system. It took approximately 170 steps. We list a sketch of the mechanized proof in Appendix.

3.4 Program Synthesis

In thie section, we describe a complete example of Constructive Programming using our system. The synthesized program is append of lists.

3.4.1 The specification of append and its proof

The specification of the append program is written as the following proposition in RPT:

$$\forall x.(\mathtt{List}[x] \supset \forall y.(\mathtt{List}[y] \supset \exists z.\mathtt{Append}[x,y,z]))$$

Here two predicates List and Append are defined as follows:

$$\begin{aligned}
\mathtt{List} &\triangleq \lambda f.\lambda x.(x = \mathtt{nil} \vee \exists x_1.\exists x_2.\ \ x = \langle x_1, x_2\rangle \wedge f[x_2]) \\
\mathtt{Append} &\triangleq \lambda f.\lambda x.\lambda y.\lambda z.\ x = \mathtt{nil}\ \wedge\ y = z \\
&\qquad \vee\ \exists x_1.\exists x_2.(x = \langle x_1, x_2\rangle \wedge\ \exists z_1.(f[x_2,y,z_1]\ \wedge\ z = \langle x_1, z_1\rangle)))
\end{aligned}$$

We proved the specification formula above using our system. It took about 80 steps including the definitions of List and Append, and several naming operation of intermediate theorems.

Let T_{app} be the proof-term of the specification above. Its precise form is as follows:

```
μ(λfxq.
   (λr.if car(r) then
         λyl. [y true 0 . 0]
       else
         λyl.
          [[cadr(r) . car(cddddr(r)yl)]
           false
           cadr(r)
           caddr(r)
           0
           car(cddddr(r)yl)
           cdr(cddddr(r)yl)
          . 0]
       fi)
   [car(q) .
     if car(q) then cdr(q)
     else
       [cadr(q)
        caddr(q)
        cadddr(q) .
        f(caddr(q))(cddddr(q))]
     fi
   ]
  )
```

The term T_{app} requires four arguments:

- List x,
- a proof q that x is a list,
- List y, and
- a proof l that y is a list.

The return value of T_{app} is a proof of $\exists z.\mathtt{Append}[x, y, z]$. Therefore, the result of append'ing two lists x and y is $\mathtt{car}(T_{app} x(r_{list}(x)) y(r_{list}(y)))$ where $r_{list}(x)$ is a proof that x is a list.

3.4.2 Improvement of extracted programs

The program T_{app} is guaranteed to be correct with respect to the specification; however it is by no means satisfactory, since it is inefficient, and it requires extra arguments other than x and y. We will discuss the first problem in the next subsection. Here we discuss the second problem.

Theorem 9 *The following holds for some term a:*

$$a \vdash_1 (\mathtt{List}[x] \supset r_{list}(x) \vdash_0 \mathtt{List}[x])$$

where r_{list} is defined as follows:

```
$r_{list} \triangleq \mu(\lambda f.\lambda x.$
        if null?($x$) then $\langle$true, 0$\rangle$
           else [false.[car($x$).[cdr($x$).$\langle 0, f($cdr($x$)$)\rangle$]]] fi) □
```

This theorem claims that a proof of listhood of x is obtained by a computation using x. Therefore, we do not need $r_{list}(x)$ besides x.

Similar theorems hold for other practically useful data types such as natural numbers, lists, and trees.

3.4.3 Eliminating Redundancy by Program Transformation

Our next goal here is to eliminate redundant parts from the naively extracted program T_{app}. We achieve it by transforming the program preserving the correctness.

The intensional equivalence relation $\approx$ given in Chapter 2 is too fine for this purpose. We need a more coarse, extensional equivalence relation.

In a call-by-value calculus, the values (results of computation) are normal terms. Then, we can define an extensional equality as follows: Let F and G be unary functions. They are extensionally equal if, for any normal terms n and m, $F(n) \rightarrow^* m$ if and only if $G(n) \rightarrow^* m$. F and G are not necessarily intensionally equal. If F and G are extensionally equal, we may replace F by G in any context.

The extensional equality of Λ is defined similarly as this extensional equality. However, our definition is more complex than this, since the computation of Λ is call-by-name rather than call-by-value, and the values are canonical terms rather than normal terms. For instance, let $\mathtt{I}$ be the following term:

$$(\mu(\lambda f.\lambda n.\langle n, f(suc(n))\rangle))0$$

where $suc(n)$ represents the successor of n. The term $\mathtt{I}$ represents an infinite list of natural numbers. Both terms Ω and $\mathtt{I}$ are not terminating, so they are equal by the above extensional equality. However, they do not necessarily have the same meaning. For example, if we substitute one of them for x in $\mathtt{pair?}(x)$, then the results are different. It follows that we need a more sophisticated definition of the extensional equality.

Here, we will introduce a new equality $\sim$ by regarding Λ as a lazy computation system in the sense of [25].

Definition 15 (Preorder in lazy computation system) *Let R be a binary relation on closed terms. Then a binary relation $T[R]$ is defined as follows:*

$$\begin{aligned}
s\ T[R]\ t \triangleq\ & \\
& (s \rightarrow^* \mathtt{nil} \supset t \rightarrow^* \mathtt{nil}) \\
& \wedge (s \rightarrow^* \mathtt{true} \supset t \rightarrow^* \mathtt{true}) \\
& \wedge (s \rightarrow^* \mathtt{false} \supset t \rightarrow^* \mathtt{false}) \\
& \wedge \forall s_1, s_2.\ (s \rightarrow^* \langle s_1, s_2\rangle \supset \exists t_1, t_2.\ (t \rightarrow^* \langle t_1, t_2\rangle \wedge (s_1\ R\ t_1) \wedge (s_2\ R\ t_2))) \\
& \wedge \forall a.\ (s \rightarrow^* \lambda x.a \supset \exists b.\ (t \rightarrow^* \lambda y.b \wedge \forall u.\ (a_x[u]\ R\ b_y[u])))
\end{aligned}$$

where s_1, s_2, t_1, t_2, u range over closed terms, and a, b range over terms[1].

A preorder $\leq$ on closed terms is the largest fixpoint of $R = T[R]$. □

In the above definition of $T[R]$, every occurrence of R is strictly positive, so the equality $R = T[R]$ has the largest fixpoint.

We can extend the preorder $\leq$ to open terms; Let $FV(F) \cup FV(G)$ be $\{x_1, \cdots, x_n\}$. Then $F \leq G$ if, for any closed terms $d_1, \cdots, d_n$, $F_{\vec{x}}[\vec{d}] \leq G_{\vec{x}}[\vec{d}]$.

As an example of $\leq$, for any term t, we have $\Omega \leq t$. We also have $\mathtt{I} \leq \langle 0, \langle x, y\rangle\rangle$.

Definition 16 (Extensional equality in lazy computation system) *Two terms F and G are extensionally equal if $F \leq G$ and $G \leq F$. We write $F \sim G$ if they are extensionally equal.* □

Theorem 10 *We have the following:*
1. $\leq$ is reflexive and transitive.
2. $a \approx b$ implies $a \leq b$.
3. $\sim$ is an equivalence relation. □

[1]In this definition, the logical connectives $\wedge$ and $\forall$ are not formal ones in RPT.

The proof of the fist clause makes use of the fact that $\leq$ is the largest fixpoint.

Following the terminology of [25], Λ is operator extensional. Hence, $\sim$ is a congruence relation, namely, $\sim$ commutes with the construction of terms. Moreover, Λ satisfies several conditions stated in [25] such as the deterministic condition, hence $\sim$ coincides with the observational equivalence[2].

In the following, we will give transformation rules which are correct with respect to the extensional equality $\sim$.

Theorem 11 *Let x, y, z, u, v, w be variables, and a, b be terms. Let* `fun` *be one of* `car`, `cdr`, `null?`, `true?`, `false?`, `pair?`, *or* `fun?`. *Then we have*

$$\texttt{fun}(\texttt{if } x \texttt{ then } y \texttt{ else } z \texttt{ fi}) \sim \texttt{if } x \texttt{ then fun}(y) \texttt{ else fun}(z) \texttt{ fi},$$

$$\begin{array}{l}
(\texttt{if } x \texttt{ then } y \texttt{ else } z \texttt{ fi})(w) \\
\quad \sim \texttt{if } x \texttt{ then } y(w) \texttt{ else } z(w) \texttt{ fi}, \\
\texttt{if } (\texttt{if } x \texttt{ then } y \texttt{ else } z \texttt{ fi}) \texttt{ then } u \texttt{ else } v \texttt{ fi} \\
\quad \sim \texttt{if } x \texttt{ then } (\texttt{if } y \texttt{ then } u \texttt{ else } v \texttt{ fi}) \\
\qquad \texttt{else } (\texttt{if } z \texttt{ then } u \texttt{ else } v \texttt{ fi}) \texttt{ fi}, \textit{ and} \\
\texttt{if } x \texttt{ then } a \texttt{ else } b \texttt{ fi} \\
\quad \sim \texttt{if } x \texttt{ then } a_x[\texttt{true}] \texttt{ else } b_x[\texttt{false}] \texttt{ fi}
\end{array}$$

□

Proof. .

We will prove the case for `car` of the first equation only.

Let θ be a substitution which substitutes closed terms for x, y, z.

$L \triangleq \texttt{car}(\texttt{if } x \texttt{ then } y \texttt{ else } z \texttt{ fi})\theta$

$M \triangleq (\texttt{if } x \texttt{ then car}(y) \texttt{ else car}(z) \texttt{ fi})\theta$

We first show $L \leq M$. If L does not have a canonical form, then $L \leq M$ trivially holds. Hence we assume L has a canonical form. By the reduction rules of Λ, we have either (i) $x\theta \to^* \texttt{true}$ and for some terms a, b, $y\theta \to^* \langle a, b\rangle$, or (ii) $x\theta \to^* \texttt{false}$ and for some terms a, b, $z\theta \to^* \langle a, b\rangle$. In either case, we have $L \to^* a$ and $M \to^* a$, hence we have $L \leq M$.

We can show $M \leq L$ similarly. □

The terms in both sides of equations in Theorem 11 are not equal in the sense of $\approx$. Hence, $F \sim G$ does not imply $F \approx G$.

We then define strict terms. Intuitively, a strict term c with respect to f is that needs the value of f in its computation.

Definition 17 (Strict Term) *For a variable f, a strict term c is defined as follows:*

$$c \ ::= \ f$$

[2]Two terms a and b are observationally equivalent if, for any context $C\langle\cdot\rangle$, $C\langle a\rangle$ and $C\langle b\rangle$ reduce to canonical forms of the same kind, or they do not reduce to canonical forms.

$$\begin{array}{ll}
| & \texttt{null?}(c) \mid \texttt{true?}(c) \mid \texttt{false?}(c) \\
| & \texttt{pair?}(c) \mid \texttt{fun?}(c) \\
| & \texttt{car}(c) \mid \texttt{cdr}(c) \\
| & c(t) \\
| & \texttt{if } c \texttt{ then } t \texttt{ else } t \texttt{ fi}
\end{array}$$

where t is a term. □

Theorem 12 *Suppose the following hold for a term F, mutually distinct variables $f, x_1, \cdots, x_n$, terms c, a, b_i^j $(1 \leq i \leq n, 1 \leq j \leq k)$:*

$$\begin{array}{l}
FV(F) = \{f\} \\
FV(a) = \emptyset \\
FV(b_i^j) \subseteq \{x_1, \cdots, x_n\} \\
FV(c) \subseteq \{f, x_1, \cdots, x_n\} \\
c \textit{ is strict with respect to } f \\
c_f[F] \sim a(c_{\vec{x}}[\vec{b^1}], \cdots, c_{\vec{x}}[\vec{b^k}], \vec{x})
\end{array}$$

Then the following holds.

$$c_f[\mu(\lambda f.F)] \sim G(\vec{x})$$

where G is defined as follows:

$$G \triangleq \mu(\lambda g.\lambda\vec{x}.a(g(\vec{b^1}), \cdots, g(\vec{b^k}), \vec{x}))$$

□

Theorem 12 plays a central role when we eliminate redundancy in recursive functions which uses μ. For instance, let F be the following term:

$\lambda x.\texttt{if null?}(x) \texttt{ then } \langle b, c\rangle$

$\qquad\qquad \texttt{else } [p(\texttt{car}(f(pred(x)))).\ q(f(pred(x)))] \texttt{ fi}$

where $pred(x)$ is the predecessor of x, and $FV(p) = \{x\}$. Let F' be $\mu(\lambda f.F)$, then apply F' to a natural number n. During the iteration of recursive calls, the intermediate values are always of the form of a pair. The pair $F'(n)$ uses only the first component of the pair $F'(pred(n))$, Therefore, if we want to have the first component of the pair, namely, $\texttt{car}(F'(n))$, then we do not have to compute the second component. Formally speaking, let H be the following:

$$\begin{array}{l}
\lambda x.\texttt{if } \ \texttt{null?}(x) \texttt{ then } b \\
\qquad\quad \texttt{else } p(g(pred(x))) \texttt{ fi}
\end{array}$$

Intuitively, we can replace $\texttt{car}(\mu(\lambda f.F)x)$ by $\mu(\lambda g.H)x$. We do not have

$$\texttt{car}(\mu(\lambda f.F)x) \approx \mu(\lambda g.H)x$$

however, we have

$$\texttt{car}(\mu(\lambda f.F)x) \sim \mu(\lambda g.H)x$$

hence this replacement is justified with respect to $\sim$.

This example is an instance of Theorem 12 where c, F and G are $\texttt{car}(fx)$, F and $\mu(\lambda g.H)$. Also we used Theorem 11 in the transformation of

$$c_f[F] \sim \texttt{if null?}(x)\ \texttt{then}\ b\ \texttt{else}\ p(\texttt{car}(f(pred(x))))\ \texttt{fi}$$

Theorem 13 *For mutually distinct variables $f, y, x_1, \cdots, x_m$, terms F, G, a, b^j_i, c^j $(1 \leq i \leq m, 1 \leq j \leq k)$, if the following holds:*

$$\begin{aligned}
&F \equiv \lambda\vec{x}.\lambda y.a(f(\vec{b^1}, c^1), \cdots, f(\vec{b^k}, c^k), \vec{x})\\
&G \equiv \mu(\lambda f.F)\\
&FV(a) = \emptyset\\
&FV(b^j_i) \subseteq \{x_1, \cdots, x_m\}\\
&FV(c^j) \subseteq \{x_1, \cdots, x_m, y\}
\end{aligned}$$

then we have

$$G(\vec{x}, y) \sim \mu(\lambda g.\lambda\vec{x}.a(g(\vec{b^1}), \cdots, g(\vec{b^k}), \vec{x}))(\vec{x})$$

□

This theorem means that, if some variables are not used during the iteration of recursive calls, then the variable (y in the above theorem) can be eliminated. Intuitively it is obvious, and it is proved similarly as Theorem 12.

Improving efficiency of the append program

Let F be the body of μ in T_{app}, and c be $\texttt{car}(fx(r_{list}(x))yl)$. Then, by using Theorem 11, the term $c_f[F]$ can be simplified, and then we have the following by Theorem 12:

$$\begin{aligned}
&\texttt{car}(T_{app}x(r_{list}(x))yl)\\
&\quad \sim \mu(\lambda g.\lambda x, y, l.\ \texttt{if null?}(x)\ \texttt{then}\ y\\
&\qquad\qquad\qquad \texttt{else}\ \langle\texttt{car}(x), g(\texttt{cdr}(x), y, l)\rangle\ \texttt{fi})
\end{aligned}$$

In the righthand side of the above equation, the variable l is not used. Hence by Theorem 13, we have:

$$\begin{aligned}
&\texttt{car}(T_{app}x(r_{list}(x))yl)\\
&\quad \sim \mu(\lambda g.\lambda x, y.\ \texttt{if null?}(x)\ \texttt{then}\ y\\
&\qquad\qquad\qquad \texttt{else}\ \langle\texttt{car}(x), g(\texttt{cdr}(x), y)\rangle\ \texttt{fi})
\end{aligned}$$

The last definition is the same as the usual hand-written program of `append`.

Discussion

The optimization technique given in this section is built-in for our Constructive Programming System. In fact, we can obtain the final `append` program from the term $\texttt{car}(T_{app}x(r_{list}(x))yl)$ completely automatically. We can therefore, obtain an efficient and correct program.

However, the correctness of the technique in this section relies upon several metatheorems such as Theorem 12. If we want to be completely formal, then we have to formalize the extensional equality $\sim$, which is defined as the largest fixpoint. It is left for future work.

In general, it is quite difficult to optimize inefficient programs automatically. However, our method can cover the cases considered in [44], and we believe that it is applicable to a wide range of programs.

3.5 Conclusion

In this chapter, we presented a Constructive Programming System based on RPT, and showed a formal proof of the Church-Rosser property in RPT. We also presented a concrete example of Constructive Programming as well as an optimization technique of a naively extracted program.

There have been proposed several computer softwares which support proof development in constructive logic; Nuprl system[10] for Martin-Löf's type theory, PX system[21] for Feferman's T_0, Coq system[14] for an extension of CoC, and others.

Compared with these existing systems, the characteristic points of our system are (1) the system is implemented by the object language Λ so that we can reason about the properties of the system itself, and (2) the underlying logic RPT has the built-in reflection mechanism, hence we can internally express metaproperties in our system. We have demonstrated the use of the reflection mechanism in our system. Recently it has been widely recognized that the reflection mechanism is quite useful in both theories and practices[6].

We plan to prove larger examples using our system so that we can extract more realistic programs. In order to do so, we will have to improve our system at two points; (1) introducing backward-reasoning, and providing various kinds of tactics, and (2) improving the user-interface including graphical user interface.

Appendix The mechanized proof of Theorem 8

In this appendix, a summary of a proof of Theorem 8 is given.

A line beginning with a semicolon (;) is a comment line, which was attached by hand. Other lines are input by a user or output by the system.

```
Result:1: |-  ∀ x.(|= S[x])
; Assume this proposition
```

```
Result:3: |-  ∀x. ∀y.(|= A[x,y])
; Assume this proposition

Result:5: |-  ∀x. ∀y.(A[x,y] ⊃ S[x] ⊃ S[y])
; Assume this proposition

Result:14: |-  ∀x.(S[x] ⊃ ∀z. ∀y.(A[x,y] ∧ A[x,z] ⊃
∃u.(A[y,u] ∧ A[z,u])))
; Assume this proposition

RPT:21> (defindpred TransA (x y) it X)
pred-name: TransA
pred-body: x=y ∨ ∃z.(A[x,z] ∧ TransA[z,y])
OK
; Define the predicate TransA

RPT:26> (name BasicProp1 (all-intro it w))
Result:26: |-  ∀w.TransA[w,w]
; Name the proof of this proposition as BasicProp1

RPT:35> (name BasicProp2 (all-intro (all-intro it y) x))
Result:35: |-  ∀x. ∀y.(A[x,y] ⊃ TransA[x,y])
; Prove x->y ⊃ x->*y

RPT:69> (defpred Prop1 (x y) it)
pred-name: Prop1
pred-body: S[x] ⊃ ∀z.(A[x,z] ⊃ ∃u.(TransA[y,u] ∧ TransA[z,u]))
OK
; Name the proof of this proposition as Prop1

RPT:80> (name IH (assume (prop-and (prop-pred A (x z))
(prop-pred Prop1 (z y))) a3))
Result:80: |-  A[x,z] ∧ Prop1[z,y]
; Induction Hypothesis in the proof of Prop1

RPT:117> (name Proof-of-Prop1)
Result:117: |-  ∀x. ∀y.(TransA[x,y] ⊃ Prop1[x,y])

RPT:123> (defpred Prop2 (x y) it)
pred-name: Prop2
pred-body: S[x] ⊃ ∀z.(TransA[x,z] ⊃ ∃u.(TransA[y,u] ∧ TransA[z,u]))
OK
```

```
; Define the above predicate Prop2

RPT:133> (name IH2 (assume (prop-and (prop-pred A (x z))
(prop-pred Prop2 (z y))) a13))
Result:133: |-  A[x,z] ∧ Prop2[z,y]
; Induction Hypothesis in the proof of Prop2

RPT:175> (name Proof-of-Prop2)
Result:175: |-  ∀x. ∀y.(TransA[x,y] ⊃ Prop2[x,y])
; the Diamond property
```

Chapter 4

Half-monotone Inductive Definitions

This chapter studies an extension of inductive definitions in the context of a type-free theory. It is a kind of simultaneous inductive definition of two predicates where the defining formulas are monotone with respect to the first predicate, but not monotone with respect to the second predicate. We call this inductive definition *half-monotone* in analogy of Allen's term *half-positive*.

We can regard this definition as a variant of monotone inductive definitions by introducing a refined order between tuples of predicates. We give a general theory for half-monotone inductive definitions in a type-free first-order logic. We then give a realizability interpretation to our theory, and prove its soundness by extending Tatsuta's technique.

The mechanism of half-monotone inductive definitions is shown to be useful in interpreting many theories, including the Logical Theory of Constructions, and Martin-Löf's Type Theory. We can also formalize the provability relation "a term p is a proof of a proposition P" naturally. As an application of this formalization, several techniques of program/proof-improvement can be formalized in our theory, and we can make use of this fact to develop programs in the paradigm of Constructive Programming. A characteristic point of our approach is that we can extract an optimization program since our theory enjoys the program extraction theorem.

4.1 Introduction

An important problem in constructive programming is that extracted programs often contains redundant parts. Namely, a naive extraction usually produces an inefficient program. Much research has been done on this topic; Subset Types[10, 30], Separation of Spec and Prop types[32], and SUIT (Singleton, Union and Intersection Types)[23] in type theories, and Diamond suit (double negation)[21], and Extended projection[43] in type-free theories. These techniques introduce new types (in type theories) or new realizability interpretations (in type-free theories) by which we can eliminate redundant parts in programs. In other words, each of these systems gives a fixed, uniform way of program improvement. In order to introduce a new technique for improvement, they must re-define the whole system and re-prove its consistency (in type theories), or re-define the realizability interpretation and re-prove its soundness (in type-free theories). These tasks belong to meta-theories, and go beyond the original theory.

Our aim is to formalize various program improvement techniques in a single framework. Namely, we want to have a mechanism to define and reason about the relation "a term a is a program (proof) of a type (proposition) A". Since this is a metatheoretic notion, our theory should include a certain kind of reflection. If such a theory is formulated, we can add a new optimization technique to the system by re-defining the relation, and prove, for instance, equivalence of the old and new definitions. Moreover, if the metatheory is also constructive in nature, we can extract an optimization program from the proof in the metatheory. The reflection mechanism is quite useful in proof/program development, as pointed out by Allen et al[6].

However, the natural definition of these relations leads us outside the realm of *positive* inductive definitions as shown below. We call a pair of the following two relations a *provability relation.*

A is a proposition (written as $\mathtt{Prop}(A)$)
a is a proof of A (written as $\mathtt{Proves}(a,A)$)

In Martin-Löf's type theory[29], these relations correspond to the judgements $A\ Set$ and $a \in A$. In a usual first-order logic, they correspond to a metamathematical statement "A is a formula" and a formula "a realizes A". If we formulate $\mathtt{Prop}(x)$ and $\mathtt{Proves}(x,y)$ naturally, they look like:

$$\begin{aligned}
\mathtt{Prop}(y) \ \leftrightarrow\ & \cdots \\
& \exists u.\exists v.\ \mathtt{Prop}(u) \wedge y = (u \dot{\supset} v) \wedge \forall e.\ (\mathtt{Proves}(e,u) \supset \mathtt{Prop}(v)) \\
& \cdots \\
\mathtt{Proves}(x,y) \ \leftrightarrow\ & \cdots \\
& \exists u.\exists v.\ \mathtt{Prop}(u) \wedge y = (u \dot{\supset} v) \\
& \qquad \wedge \forall e.\ (\mathtt{Proves}(e,u) \supset \mathtt{Proves}(x(e),v)) \\
& \cdots
\end{aligned}$$

where $\dot{\supset}$ represents a constant corresponding to logical implication, and $\leftrightarrow$ represents logical equivalence. In order to regard these clauses as an instance of simultaneous inductive definitions for two predicates, the right hand side of $\leftrightarrow$ should be *monotone*. However, in the above formulation, an occurrence of $\mathtt{Proves}(e,u)$ appears in the left side of $\supset$, which means that it is a negative occurrence. This

kind of inductive definitions does not fall in an ordinary scheme of *positive* inductive definitions.

Several researchers have attacked this problem in different contexts.

Beeson[7] used two kinds of techniques; the first one uses the ordinal numbers in classical set theory. The other one simultaneously defines a pair of the predicates $\mathtt{Proves}$ and its negation $\overline{\mathtt{Proves}}$ by a monotone inductive definition, and then $\neg\mathtt{Proves}(x,y) \leftrightarrow \overline{\mathtt{Proves}}$ is proved using the law of excluded middle. Both techniques rely on classical logic.

Aczel[2] proposed Frege structures. Frege structures have two basic notions, "a term a is a proposition" and "a term a is a true proposition". In order to construct Frege structures in set theory, he encountered similar difficulty. As a solution, he introduced a new order between a pair of predicates. With respect to this order, the pair of defining formulas of the two notions above becomes monotone, therefore the least fixpoint exists. Although his metatheory was classical, he stated that the construction might be considered constructive. Hayashi and Nakano[21] used a similar order to construct models for their theory PX. Our method is similar to their works in that we regard our inductive definitions as monotone by changing the definition of the order. What is new in our theory is that we give a general form of such a style of inductive definitions, and that we do everything in a constructive framework in the sense that our theory has a sound realizability interpretation, and enjoys the program extraction theorem. In this chapter, we will interpret a formalized version of Frege structures (the Logical Theory of Constructions[3]) in our theory.

Allen[4, 5] gave a type-free interpretation for Martin-Löf's type theory. He encountered a similar situation as ours, and called the necessary scheme of inductive definitions *half-positive*. By using an inductive definition for higher order predicates, he was able to use ordinary (strictly positive) inductive definitions to interpret Martin-Löf's type theory. His arguments can be understood by classical set theorists as well as constructivists who accept monotone inductive definitions. Although his motivation was different from ours, we shall compare his work and ours in Section 6. Smith[40] formalized Allen's interpretation in a Martin-Löf's type theory plus recursive types.

Dybjer[15, 16] presented a general formulation of a recursive-induction mechanism so that the universe hierarchy becomes definable in Martin-Löf-style type theory. The motivation of our theory is similar to his approach, and we shall also compare his work and ours in Section 6.

In this chapter we present a general mechanism of inductive definitions by which one can define the provability relation and similar notions naturally. The mechanism is based on a form of a monotone inductive definition with a refined order between predicates. We call this mechanism *half-monotone* inductive definitions. We define a realizability interpretation to our theory, and prove its soundness by following Tatsuta's work[46]. We shall demonstrate how various concepts can be defined in our theory; in particular, we show that the Logical Theory of Constructions and

Martin-Löf's type theory can be interpreted. We also show that we can improve the efficiency of programs by defining a refined provability relation. Our theory enjoys a sound realizability interpretation, so that we have the program extraction theorem. By this theorem, we can extract programs from proofs of formalized metatheorems.

This chapter is organized as follows. Section 4.2 gives our basic theory **BT**. Section 4.3 introduces our inductive definition mechanisms, **HMID** and **HMID**$_0$ into **BT**, and gives its model. Section 4.4 presents a realizability interpretation, and proves soundness for a restricted version **BT+HMID**$_0$. Section 4.5 formalizes provability relations in our theory. We also show that program improvement is possible in **BT+HMID**$_0$. Section 4.6 interprets the Logical Theory of Constructions and briefly mentions the interpretation of Martin-Löf's type theory in **BT+HMID**$_0$. Section 6 compares our work with Allen's and Dybjer's works, and gives concluding remarks.

4.2 The basic theory

We define our basic theory **BT** in this section.

BT is an intuitionistic first-order theory for computation. Although our final goal is to formalize Frege structures[2], Reflective Proof Theory[34] and similar theories, we shall give our basic results in the framework of ordinary first-order logic, since it is easier to understand.

The definition of terms in our language is a slight modification of Sato's[34]. Terms are essentially type-free λ-terms with α and β conversions. We adopt the call-by-name semantics, but the material of this chapter can easily be applied to the call-by-value semantics.

We assume that there is a countably infinite set of variables and a finite set of constants, and that `nil`, `true`, and `false` are constants. We also assume that a unary function symbol **c?** is uniquely associated with each constant **c**.

Definition 18 (Term)

$$\begin{array}{rcl} t & ::= & x \\ & | & \mathbf{c} \mid \mathbf{c?}(t) \\ & | & \langle t_1, t_2\rangle \mid \mathtt{car}(t) \mid \mathtt{cdr}(t) \mid \mathtt{pair?}(t) \\ & | & \lambda x.t \mid t_1(t_2) \mid \mathtt{fun?}(t) \\ & | & \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3\ \mathtt{fi} \end{array}$$

where x is a variable and **c** *is a constant.*

Terms $\langle t_1, t_2\rangle$, $\mathtt{car}(t)$, and $\mathtt{cdr}(t)$ correspond to `cons`, `car`, and `cdr` in Lisp. We use the standard abbreviation for lists; $\langle t_1\rangle$ for $\langle t_1, \mathtt{nil}\rangle$, and $\langle t_1\ t_2\rangle$ for $\langle t_1, \langle t_2, \mathtt{nil}\rangle\rangle$ and so on. Terms $\lambda x.t$, $t(t)$ and $(\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3\ \mathtt{fi})$ are as usual. Terms of the form of **c**, $\langle t_1, t_2\rangle$, or $\lambda x.t$ are called *canonical*. For a canonical term t, the terms

$\mathbf{c?}(t)$, $\mathtt{pair?}(t)$ and $\mathtt{fun?}(t)$ are equal to `true` if they are in either form of $\mathbf{c?}(\mathbf{c})$, $\mathtt{pair?}(\langle s, u\rangle)$ or $\mathtt{fun?}(\lambda x.s)$, and equal to `false` otherwise. We assume a call-by-name evaluation mechanism; for example, $\mathtt{car}(\langle \mathtt{nil}, t\rangle)$ is equal to `nil` regardless of t. A term of the form $a_x[t]$ denotes the result of substitution as usual.

We then define formulas and abstracts. We assume, for each natural number n, there is a countably infinite set of predicate variables with arity n. Predicate variables are used for inductive definition. Since they are not quantified by $\forall$ or $\exists$, **BT** is a first-order theory.

Definition 19 (Formula and n-ary Abstract)

$$\begin{array}{rcl} F & ::= & \bot \mid t_1 = t_2 \mid t\downarrow \\ & | & F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \supset F_2 \\ & | & \forall x.F \mid \exists x.F \\ & | & A^n(t_1, \cdots, t_n) \\ A^0 & ::= & F \\ A^n & := & X^n \\ & | & \lambda x.A^{n-1} \\ & | & \mu X^n.A^n \\ & & (\textit{for each } n > 0) \end{array}$$

where x, t and X^n are meta variables for a variable, a term, and an n-ary predicate variable.

A formula of the form $\bot$, $t_1 = t_2$, or $t\downarrow$ is called *atomic*. $a\downarrow$ means that the term a has a canonical form. The formula $A^n(t_1, \cdots, t_n)$ represents the application of n terms $t_1, \cdots, t_n$ to an abstract A^n. The superscript n for predicate variables and abstracts is often omitted. We sometimes call an abstract a *predicate*. We write a list of n distinct variables $x_1, \cdots, x_n$ as **x**, and a list of n terms $t_1, \cdots, t_n$ as **t**. Similarly, the formula $A(t_1, \cdots, t_n)$ is abbreviated as $A(\mathbf{t})$. $\neg A$ and $A \leftrightarrow B$ are abbreviations for $A \supset\bot$ and $(A \supset B) \wedge (B \supset A)$, respectively. The formula $A_X[B]$ represents the formula A with B substituted for X. We often omit the subscript X if it is apparent from the context. It is also written as $A\{X := B\}$.

In the following, we will use meta-variables x, y, z, w for variables, s, t, u for terms, A, B, F, G for formulas and abstracts, and X, Y for predicate variables. The precedence of connectives is in the order $\wedge, \vee, \supset, \forall, \exists$ (the former is stronger). We also assume $\supset$ is right associative, namely $A \supset B \supset C$ means $A \supset (B \supset C)$.

The formal system of **BT** is given in the natural deduction style. We have the following three classes of axioms and inference rules:

- Axioms and Rules for terms

- Rules for equality
- Rules for logical connectives

These axioms and rules are quite standard for first-order intuitionistic logic, and therefore omitted. We do not adopt a logic of partial terms, so quantifiers range over arbitrary terms, not necessarily canonical nor normal terms.

If a formula F is proved from a set of formulas $A_1, \cdots, A_n$ in a theory $\mathcal{T}$, we will write it as $A_1, \cdots, A_n \vdash_{\mathcal{T}} F$.

4.3 Inductive Definitions

In this section, we present an extended mechanism of simultaneous inductive definition, which plays a central role in our theory.

As we explained in Section 4.1, we need a simultaneous definition of two predicates `Prop` and `Proves`. The definition should allow negative occurrences of predicate variables in the template[1], which means the template may sometimes be non-monotonic. This kind of definitions is not legal in conventional formal theories for positive inductive definitions.

However, the intuitive meaning of the pair $\langle$`Prop`, `Proves`$\rangle$ has some kind of monotonicity; `Prop`$(A \dot\wedge B)$ is defined by using `Prop`(A) and `Prop`(B), which are already defined. `Prop`$(A \dot\supset B)$ is defined by using `Prop`(A), `Prop`(B), and `Proves`(x, A). Since the set of those r satisfying `Proves`(r, A) is defined at the time of defining `Prop`(A) and will never be changed after defined, we may consider that `Prop`$(A \dot\supset B)$ is defined by using already defined concepts only.

To formalize this idea, we define a refined order $\leq_R$ on a pair of unary and binary predicates $\langle P, Q\rangle$ as follows:

$$\begin{aligned}\langle P_0, Q_0\rangle \leq_R \langle P_1, Q_1\rangle \triangleq\ & \forall x.\ (P_0(x) \supset P_1(x)) \ \wedge\ \forall x, y.\ (Q_0(x,y) \supset Q_1(x,y)) \\ & \wedge\ \forall x.\ (P_0(x) \supset \forall y.(Q_0(x,y) \leftrightarrow Q_1(x,y)))\end{aligned}$$

It is essential to have two predicates which have an *overlapping* argument. The second predicate Q_1 can be larger than Q_0 only outside of the domain of P_0, which means that those y satisfying $Q(x,y)$ is determined at the time $P(x)$ becomes true. With respect to $\leq_R$, the naive definition of $\langle$`Prop`, `Proves`$\rangle$ becomes monotone, and we can regard it as a legal inductive definition. We call a monotone inductive definition wrt $\leq_R$ a *half-monotone* inductive definition following Allen's term *half-positive*[5].

Aczel[2] used a similar order $\leq_A$ in his semantical framework. For two sets S and T with $T \subset S$, the order $\leq_A$ is defined as follows:

$$\langle S_0, T_0\rangle \leq_A \langle S_1, T_1\rangle \triangleq S_0 \subset S_1 \wedge \forall x \in S_0.\ (x \in T_0 \leftrightarrow x \in T_1)$$

It is easily seen that $\leq_A$ is a special case of $\leq_R$: For a pair of sets $\langle S_i, T_i\rangle$, we define $P_i(x) \triangleq x \in S_i$ and $Q_i(x,y) \triangleq x \in T_i \wedge y = 0$. Then $\langle S_0, T_0\rangle \leq_A \langle S_1, T_1\rangle$ are

[1] We call the defining formula in an inductive definition a *template*.

equivalent to $\langle P_0, Q_0\rangle \leq_R \langle P_1, Q_1\rangle$ provided that $T_i \subset S_i$ (for $i = 0, 1$) holds. Hence, our order can be regarded as a generalization of Aczel's.

In this section, we first review Tatsuta's theory and realizability interpretation for ordinary monotone inductive definition in a context of a type-free first-order theory. We then introduce a half-monotone inductive definition and construct a model of the extended theory. A realizability interpretation and its soundness proof will be given in the next section.

4.3.1 Monotone inductive definition

Tatsuta[46] introduced into Beeson's **EON**[7] a mechanism of monotone inductive definitions. Then he defined a q-realizability interpretation, and proved its soundness for a restricted version. We first reformulate his results using our theory **BT** instead of **EON**.

Definition 20 (Natural order between abstracts) *Let P_0, P_1 be n-ary abstracts, and* $\mathbf{x}$ *be a list of n distinct variables. Then we define*

$$P_0 \leq_N P_1 \ \triangleq\ \forall \mathbf{x}.P_0(\mathbf{x}) \supset P_1(\mathbf{x})$$

This relation is an order *modulo logical equivalence*, namely we identify two predicates which are logically equivalent. The subscript N indicates this order is a natural one.

Definition 21 (Monotonicity wrt. $\leq_N$) *Let A be an n-ary abstract, which possibly contains an n-ary predicate variable X. Then we define*

$$\mathbf{MONO}(A; X) \ \triangleq\ X_0 \leq_N X_1 \ \supset\ A_X[X_0] \leq_N A_X[X_1]$$

where X_0 and X_1 are fresh n-ary predicate variables.

We often omit X in $\mathbf{MONO}(A; X)$. We say A is monotone in X if $\mathbf{MONO}(A; X)$ holds.

The mechanism of *monotone inductive definitions* (**MID**, in short) is that, for any A satisfying $\mathbf{MONO}(A; X)$, we have the least solution of $\forall \mathbf{x}.(A(\mathbf{x}) \leftrightarrow X(\mathbf{x}))$. We denote this solution as $\mu X.A$. Since there was a technical difficulty in attaching realizers to the full **MID** in a first-order theory, Tatsuta formulated an additional condition called **MONO-Q**, and allowed **MID** only when $\mathbf{MONO\text{-}Q}(A; X)$ is satisfied. Roughly speaking, $\mathbf{MONO\text{-}Q}(A; X)$ is the condition that the relation "e realizes $A(\mathbf{x})$" is monotone in the relation "e realizes $X(\mathbf{x})$". Since $\mathbf{MONO}(A; X)$ does not subsume $\mathbf{MONO\text{-}Q}(A; X)$, this is a proper restriction.

The rules for $\mathbf{MID}_0$, a restricted version of **MID**, are as follows.

$$\frac{\mathbf{MONO}(A) \quad \mathbf{MONO\text{-}Q}(A)}{\forall \mathbf{x}.\ A_X[\mu X.A](\mathbf{x}) \leftrightarrow (\mu X.A)(\mathbf{x})}\ (\mu\text{-eq})$$

$$\frac{\mathbf{MONO}(A) \quad \mathbf{MONO\text{-}Q}(A) \quad A_X[C] \leq_N C}{\mu X.A \leq_N C} \ (\mu\text{-ind})$$

where C is an arbitrary n-ary abstract. The first rule states that $\mu X.A$ is a solution of $\forall \mathbf{x}.\ A(\mathbf{x}) \leftrightarrow X(\mathbf{x})$. The second rule states that $\mu X.A$ is a least solution. Tatsuta gave a realizability interpretation of the theory with $\mathbf{MID}_0$ in the theory with **MID**, and proved its soundness.

MID and $\mathbf{MID}_0$ can be extended to a simultaneous inductive definition straightforwardly.

Remarks

$\mathbf{MID}_0$ is an extension of positive inductive definitions (where all the occurrences of X in A must be in syntactically positive positions), since every positive A satisfies $\mathbf{MONO}(A)$ and $\mathbf{MONO\text{-}Q}(A)$. Moreover, the extension is proper as shown by Tatsuta's example, $\mu X.\ \bot \wedge (X \supset X)$, which is not positive, but trivially satisfies **MONO** and **MONO-Q** conditions. However, he has shown no substantial example which demonstrates the difference between positive and monotone inductive definitions. Therefore, practical use of his results was not known. In the following, we will extend his **MID** ($\mathbf{MID}_0$) to obtain **HMID** ($\mathbf{HMID}_0$, resplectively). Our definition of the realizability interpretation is essentially due to Tatsuta's. What is new is that our theory with $\mathbf{HMID}_0$ has many uses in defining notions such as the provability relation.

Tatsuta also gave a realizability interpretation to full **MID** in a second-order theory. This interpretation is quite straightforward, since one can directly define the least solution of $\forall \mathbf{x}.\ A(\mathbf{x}) \leftrightarrow X(\mathbf{x})$ as $\forall X.\ (\forall \mathbf{y}.A(\mathbf{y}) \supset X(\mathbf{y})) \supset X(\mathbf{x})$ in a second-order theory. He also discussed the difference of these two interpretations. Here, we do not get into the long-standing debate between first-order theories plus induction principles (intuitionistic mu-calculus) versus second-order theories with the above encoding of the least fixpoint. We just quote Tatsuta's remark that a realizer of **MID** in a first-order theory has a loop structure (a recursive call) inside it, while one in a second-order theory (via the above encoding) does not have a loop structure. See [46] for details.

4.3.2 Refined order

Here we define the refined order discussed at the beginning of this section, and will use it in inductive definitions.

Definition 22 (Refined order[2]) *Let $\langle P_0, Q_0\rangle$ and $\langle P_1, Q_1\rangle$ be pairs of predicates with arities n and $n+m$. Then we define $\langle P_0, Q_0\rangle \leq_R \langle P_1, Q_1\rangle$ as the conjunction of the following three formulas.*

[2]This relation is also an order *modulo logical equivalence*.

1. $\forall \mathbf{x}.\ P_0(\mathbf{x}) \supset P_1(\mathbf{x})$
2. $\forall \mathbf{x}, \mathbf{y}.\ Q_0(\mathbf{x}, \mathbf{y}) \supset Q_1(\mathbf{x}, \mathbf{y})$
3. $\forall \mathbf{x}, \mathbf{y}.\ P_0(\mathbf{x}) \supset Q_1(\mathbf{x}, \mathbf{y}) \supset Q_0(\mathbf{x}, \mathbf{y})$

where $\mathbf{x}$ ($\mathbf{y}$) are sequences of n (m) distinct variables.

The first two conditions are the same as the conditions for the natural order $\leq_N$. The difference arises in the third one. This condition together with the second condition says, for any $\mathbf{x}$ satisfying $P_0(\mathbf{x})$, the predicates $Q_0(\mathbf{x}, \mathbf{y})$ and $Q_1(\mathbf{x}, \mathbf{y})$ are equivalent. In other words, Q_1 is larger than Q_0 only outside of the domain (defined by P_0). We call this order a refined order or an R-order, in short. We say that the second predicate Q_i depends on the first predicate P_i.

We can extend the R-order to the case of more than 2 predicates, for instance, Q_i depends on P_i, and R_i depends on Q_i. To formalize a general scheme, we introduce a notion of a dependency function. For a natural number n, a dependency function df is a finite function from $\{1, \cdots, n\}$ to the set of natural numbers, which satisfies $0 \leq df(i) < i$. We say n is the degree of df.

Definition 23 (Refined order in general case) *Let df be a dependency function of degree n. For two sequences of predicates $\langle P^1, \cdots, P^n\rangle$ and $\langle Q^1, \cdots, Q^n\rangle$ whose arities correspond, we define $\langle P^1, \cdots, P^n\rangle \leq_{R(n,df)} \langle Q^1, \cdots, Q^n\rangle$ as:*

$$\langle P^{df(i)}, P^i\rangle \leq_R \langle Q^{df(i)}, Q^i\rangle \text{ holds for each } i\ (1 \leq i \leq n)$$

where P^0 and Q^0 are $\lambda \mathbf{x}.\ \bot$.

If $df(i) = 0$, the condition becomes $P^i \leq_N Q^i$. Note that, the orders $\leq_N$ and $\leq_R$ can be written as $\leq_{R(1,df_N)}$ and $\leq_{R(2,df_R)}$ where $df_N(1) = 0, df_R(1) = 0, df_R(2) = 1$.

Example

Let df be a dependency function of degree 4 with $df(1) = df(2) = 0, df(3) = 1$ and $df(4) = 3$. Then the order $\langle P_0, Q_0, R_0, S_0\rangle \leq_{R(4,df)} \langle P_1, Q_1, R_1, S_1\rangle$ is

$$P_0 \leq_N P_1 \ \wedge\ Q_0 \leq_N Q_1 \ \wedge\ \langle P_0, R_0\rangle \leq_R \langle P_1, R_1\rangle \ \wedge\ \langle R_0, S_0\rangle \leq_R \langle R_1, S_1\rangle.$$

We call this order $R4$-order.

4.3.3 Half-monotone inductive definition

We first extend the language of our theory. Let A_i and X_i be an n_i-ary abstract and an n_i-ary predicate variable for $1 \leq i \leq n$. Then, $\mu X_i.\langle A_1, \cdots, A_n; X_1, \cdots, X_n\rangle$ is an n_i-ary abstract with $X_1, \cdots, X_n$ bound by this μ operator. This form will be used for representing the i-th component of a least fixpoint.

The definition of monotonicity does not change, but we present it here for completeness.

Definition 24 (Monotonicity wrt. $\leq_R$) *Let $\langle A, B\rangle$ be a pair of n-ary and $n+m$-ary abstracts, each of which possibly contains n-ary and $n+m$-ary predicate variables X and Y. Then we define*

$$\begin{aligned}
&\mathbf{HMONO}(A, B; X, Y) \triangleq \\
&\quad \langle X_0, Y_0\rangle \leq_R \langle X_1, Y_1\rangle \supset \\
&\qquad \langle A_{X,Y}[X_0, Y_0], B_{X,Y}[X_0, Y_0]\rangle \leq_R \langle A_{X,Y}[X_1, Y_1], B_{X,Y}[X_1, Y_1]\rangle
\end{aligned}$$

We often omit X and Y in **HMONO**$(A, B; X, Y)$. We may similarly define the **HMONO** condition for the case of the generalized order $\leq_{R(n,df)}$.

Let us abbreviate as follows:

$$\begin{gathered}
M_X \triangleq \mu X.\langle A, B; X, Y\rangle \\
M_Y \triangleq \mu Y.\langle A, B; X, Y\rangle \\
\mu - \mathsf{eq} \triangleq \forall \mathbf{x}.\ (M_X(\mathbf{x}) \leftrightarrow A[M_X, M_Y](\mathbf{x})) \wedge \forall \mathbf{xy}.\ (M_Y(\mathbf{x}, \mathbf{y}) \leftrightarrow B[M_X, M_Y](\mathbf{x}, \mathbf{y}))
\end{gathered}$$

Then the inference rules for a half-monotone inductive definition are as follows:

$$\frac{\mathbf{HMONO}(A, B)}{\mu\text{-}\mathsf{eq}}\ (\mu\texttt{-eq-0})$$

$$\frac{\mathbf{HMONO}(A, B) \quad \langle A[F, G], B[F, G]\rangle \leq_R \langle F, G\rangle}{\langle M_X, M_Y\rangle \leq_R \langle F, G\rangle}\ (\mu\texttt{-ind-0})$$

where F and G are arbitrary n-ary and $n+m$-ary predicates.

As usual, the first rule states $\langle M_X, M_Y\rangle$ is a solution of an equation, and the second rule states the pair $\langle M_X, M_Y\rangle$ is the least solution with respect to the order $\leq_R$.

Similarly, we can define the μ-eq-0 and μ-ind-0 rules with respect to an arbitrary order $\leq_{R(n,df)}$.

The union of μ-eq-0 and μ-ind-0 for an arbitrary order $\leq_{R(n,df)}$ is called **HMID**. Since the order $\leq_{R(n,df)}$ is an extension of $\leq_N$, **HMID** is an extension of **MID**.

4.3.4 Restricted version of half-monotone inductive definition

As in **MID**, we need additional conditions for giving a sound realizability interpretation. The first condition **HMONO-Q** is a corresponding condition to **MONO-Q** in **MID**$_0$. The second one is that, we can use the μ-ind rule only when the abstract G is the least fixpoint M_Y, and the order is $\leq_R$ (of degree 2). Under these conditions, the rules become:

$$\frac{\mathbf{HMONO}(A, B) \quad \mathbf{HMONO\text{-}Q}(A, B)}{\mu\text{-}\mathsf{eq}}\ (\mu\texttt{-eq-1})$$

$$\frac{\begin{array}{l}\mathbf{HMONO\text{-}Q}(A, B) \\ \mathbf{HMONO}(A, B) \quad \langle A[F, M_Y], B[F, M_Y]\rangle \leq_R \langle F, M_Y\rangle\end{array}}{M_X \leq_N F}\ (\mu\texttt{-ind-1})$$

In the latter rule, we write two conditions horizontally for a typographic reason.

The precise form of **HMONO-Q**(A,B) is described in Section 4.4; here we just remark that **HMONO-Q**(A, B) becomes **MONO-Q**(A) when B is identically true.

Under the conditions **HMONO** and **HMONO-Q**, the first rule states that $\langle M_X, M_Y\rangle$ is a solution, and the second one states that it is the least one. Since μ-ind-1 restricts the use of induction principle only for the first predicate (F in the above definition), we cannot infer the minimality of M_Y.

Let us consider that the first predicate M_X defines a domain, and the second predicate M_Y defines a property on that domain. Then we need an induction for M_X, but not for M_Y. We have no interest in the behavior of $M_Y(\mathbf{x}, \mathbf{y})$ where $M_X(\mathbf{x})$ does not hold, since such an $\mathbf{x}$ is outside of the domain. In Sections 4.5 and 4.6, we shall show the above formulation is sufficient for most purposes.

The $\mu\texttt{-ind-1}$ rule subsumes the following more convenient form:

$$\frac{\mathbf{HMONO}(A, B)\ \mathbf{HMONO\text{-}Q}(A, B)\ A[F, M_Y] \leq_N F\ F \leq_N M_X}{M_X \leq_N F}$$

The union of the μ-eq-1 and μ-ind-1 rules (for $\leq_R$ order) is called **HMID**$_0$. Note that **HMID**$_0$ is an extension of **MID**$_0$.

Remark

Note that, for our purposes, a positive inductive definition is worthless since we cannot have a meaningful syntactic condition which subsumes the third condition in the definition of $\leq_R$:

$$\forall x, y.\ P_0(x) \supset Q_1(x, y) \supset Q_0(x, y)$$

This contrasts to the corresponding situation in type theory by Dybjer[16]. In his theory, the condition for the inductive definition is given by a purely syntactic way. We shall compare his theory and ours in the conclusion.

4.3.5 A model of HMID

Let us explain how to construct a model of **BT+HMID**.

We begin with an arbitrary model of **BT**; there is a domain D, and all the terms are interpreted as elements of D. An n-ary predicate is interpreted as a subset of D^n. Let $\mathcal{P}(S)$ be the powerset of S. Given a degree n, a dependency function df and a sequence of arities $m_1, \cdots, m_n$, we define an order $\sqsubseteq_{n,df}$ on *Dom* as follows, where *Dom* is $\mathcal{P}(D^{m_1}) \times \cdots \times \mathcal{P}(D^{m_n})$.

Definition 25 (Order $\sqsubseteq_{n,df}$) *For $S_i \subseteq D^{m_i}$, and $T_i \subseteq D^{m_i}$ $(1 \leq i \leq n)$, we define $\langle S_1, \cdots, S_n\rangle \sqsubseteq_{n,df} \langle T_1, \cdots, T_n\rangle$ if, for each i with $1 \leq i \leq n$, the conjunction of the following conditions hold.*

1. $S_{df(i)} \subseteq T_{df(i)}$
2. $S_i \subseteq T_i$
3. $T_i|S_{df(i)} \ = \ S_i|S_{df(i)}$

where $T|S \triangleq \{\langle \mathbf{s}, \mathbf{t}\rangle \mid \mathbf{s} \in S \wedge \langle \mathbf{s}, \mathbf{t}\rangle \in T\}$

The order $\sqsubseteq_{n,df}$ just corresponds to the $R(n, df)$-order. We may sometimes omit the subscript in $\sqsubseteq_{n,df}$.

Definition 26 (Monotone Operator) *A sequence of functions $F_i : Dom \to \mathcal{P}(D^{m_i})$ (for $1 \leq i \leq n$) is monotone if and only if, for all $S_i, T_i \subseteq D^{m_i}$,*

$\langle S_1, \cdots, S_n\rangle \sqsubseteq_{n,df} \langle T_1, \cdots, T_n\rangle$ *implies*

$$\langle F_1(S_1, \cdots, S_n), \cdots, F_n(S_1, \cdots, S_n)\rangle \sqsubseteq_{n,df} \langle F_1(T_1, \cdots, T_n), \cdots, F_n(T_1, \cdots, T_n)\rangle$$

Lemma 3 *The following holds:*

1. *$\sqsubseteq$ is an order on Dom.*
2. *The least element exists in Dom. Moreover, every directed subset of Dom has a least upper bound (lub) wrt $\sqsubseteq$.*
3. *There exists a least fixpoint for each monotone operator wrt $\sqsubseteq$.*

Proof. The first clause of the lemma is straightforward. The second clause means that *Dom* with $\sqsubseteq$ is a complete partial order (cpo), which implies the third clause (in classical set theory).

As for the second clause: let Δ be a directed set in *Dom*. Let L_i be $\cup\{\mathbf{X}_i \mid \langle \mathbf{X}_1, \cdots, \mathbf{X}_n\rangle \in \Delta\}$. We will prove that $\langle L_1, \cdots, L_n\rangle$ is the lub of Δ.

Suppose it is not an upper bound of Δ. Then there exists $\langle S_1, \cdots, S_n\rangle \in \Delta$ such that $\langle S_1, \cdots, S_n\rangle \sqsubseteq \langle L_1, \cdots, L_n\rangle$ does not hold. From the definition of L_i, we have that $L_i|S_{df(i)}$ is properly larger than $S_i|S_{df(i)}$ for some i. Then we have that there exists $\langle T_1, \cdots, T_n\rangle \in \Delta$ such that $T_i|\{\mathbf{s}\}$ is properly larger than $S_i|\{\mathbf{s}\}$ for some i and some $\mathbf{s} \in S_{df(i)}$. But this contradicts to the directedness of Δ, since we do not have an upper bound of $\langle S_1, \cdots, S_n\rangle$ and $\langle T_1, \cdots, T_n\rangle$. The minimality is apparent from the definition.

We can also prove that $\langle \emptyset, \cdots, \emptyset\rangle$ is the least element in *Dom*.

In contrary to the set inclusion order, a least upper bound for an arbitrary subset wrt $\sqsubseteq_{R(n,df)}$ does not always exist, since the union of two sets may have too many

elements. In particular, the whole domain $\langle D^{m_1}, \cdots, D^{m_n}\rangle$ is not always greater than an arbitrary element in Dom.

We can interpret a monotone operator in **BT+HMID** by a monotone operator in the model. From the lemma, we can interpret each inductively defined predicate as a least fixpoint of the corresponding operator. It is easy to see the rules μ-eq-0 and μ-ind-0 are satisfied. As a consequence, we have the following theorem.

Theorem 14 (Consistency)
1. We can construct a model of **BT+HMID** *from any model of* **BT**.
2. **BT+HMID** *(therefore,* **BT+HMID**$_0$*) is consistent.*

Our model construction is based on classical set theory to ensure the existence of a least fixpoint for each monotone operator. Hence it does not differ so much from Aczel's[2] and Hayashi and Nakano's[21]. The aim of this section is just to show the consistency of our theory in its general form. Our claim that our theory is constructive in nature is not due to this model construction, but due to the realizability interpretation defined in the next section.

4.3.6 Example of HMID

Natural Numbers

`Nat`, the predicate for natural numbers, is defined as follows. Let A be the following template:

$$\lambda x.\ x = 0 \vee \exists y.\ (y = \mathtt{suc}(x) \wedge X(y))$$

where $\mathtt{suc}(x)$ is encoded by $\langle 0, x\rangle$. Then, `Nat` can be defined as $\mu X.A$. Since A is positive in X, **HMONO** and **HMONO-Q** conditions are automatically satisfied. The usual induction principle follows from μ-ind-1.

Prime Numbers

The next example is a predicate for prime numbers.

At first thought, it is defined by:

$$\mathtt{Prime}(n) \leftrightarrow \mathtt{Nat}(n) \ \wedge \ 1 < n \ \wedge \ \forall m.\ (1 < m < n \supset \neg\, \mathtt{Div}(m, n))$$

where $\mathtt{Div}(m, n)$ means "m divides n".

At the next stage, we want to replace it by the following one:

$$\mathtt{Prime}(n) \leftrightarrow \mathtt{Nat}(n) \ \wedge \ 1 < n \ \wedge \ \forall m < n. (\mathtt{Prime}(m) \supset \neg\, \mathtt{Div}(m, n))$$

Since the latter contains a negative occurrence of `Prime`, it is not directly acceptable by conventional theories.

On the contrary, we will show that we can define it by **HMID** as follows.

$$
\begin{aligned}
A &\triangleq \lambda n.\ n = 0 \ \vee\ \exists m.\ n = \mathtt{suc}(m) \wedge X(m) \\
B &\triangleq \lambda n.\ X(n) \ \wedge\ 1 < n \\
&\qquad \wedge\ \forall m < n.\ (X(m) \wedge (Y(m) \supset \neg\, \mathtt{Div}(m, n)))
\end{aligned}
$$

In this example, the argument n is the overlapping argument for A and B. We assume here that $<$ and `Div` are defined in a standard way. The second one $B(n)$ is intended to mean that n is a prime number.

The defining formula for A is the same as one for `Nat`, so one may think it is unnecessary. However, this is the trick to use the **HMID** mechanism. In this definition, $A(n)$ determines a *domain*, and $B(n)$ determines a prime number in the domain. The domain gradually increases by repeated application of A. We have to use dependency on the argument n between two predicates in order to guarantee the monotonicity with respect to the refined order.

For these A and B, the **HMONO**(A, B) and **HMONO-Q**(A, B) conditions hold as shown below. Hence, we have the least fixpoint $\langle M_X, M_Y \rangle$ for the equation $\langle A(x), B(x) \rangle \leftrightarrow \langle X(x), Y(x) \rangle$. We define `Prime` as M_Y. Using the induction rule, $M_X(x) \leftrightarrow \mathtt{Nat}(x)$ is easily proved, and we get

$$
\mathtt{Prime}(n) \ \leftrightarrow\ \mathtt{Nat}(n) \ \wedge\ 1 < n \ \wedge\ \forall m < n.(\mathtt{Prime}(m) \supset \neg\, \mathtt{Div}(m, n))
$$

as a desired result.

Verification of the HMONO(A, B) condition

Suppose that $\langle X_0, Y_0 \rangle \leq_R \langle X_1, Y_1 \rangle$ holds. Since the defining formula for A contains only a positive occurrence of X, it is easily seen $A_X[X_0] \leq_N A_X[X_1]$. From the assumption, we have $\forall z.\ (X_0(z) \supset (Y_0(z) \leftrightarrow Y_1(z)))$. Hence $B_{X,Y}[X_0, Y_0]$ is equivalent to $B_{X,Y}[X_0, Y_1]$, and we have $B_{X,Y}[X_0, Y_0] \leq_N B_{X,Y}[X_1, Y_1]$. We also have, for an arbitrary m such that $X_0(m)$ holds, $B_{X,Y}[X_0, Y_0](m) \leftrightarrow B_{X,Y}[X_1, Y_1](m)$, and we have the conclusion.

The verification of the **HMONO-Q**(A, B) condition is postponed until Section 4.4.3.

4.4 Realizability interpretation and its soundness for BT+HMID$_0$

In this section, we define a realizability interpretation for **BT+HMID**$_0$ in **BT+HMID**, and prove its soundness. We use so called **q**-realizability interpretation[7] instead of **r**-realizability. The following interpretation is essentially the same as Tatsuta's one except that we use total term logic instead of partial term logic, and that the inductive definition is extended to the R-order.

4.4.1 Realizability Interpretation

We assume that, with each predicate variable X of arity n, a predicate variable X^* of arity $n+1$ is uniquely associated. X^* will be used for representing the abstract $\lambda x e.\ e\ \mathbf{q}\ X(\mathbf{x})$.

Given a term e and a formula A in **BT+HMID**$_0$, the $\mathbf{q}$-realizability $e\ \mathbf{q}\ A$ is a formula which is defined by induction on the structure of A as follows:

1. $e\ \mathbf{q}\ A \triangleq e = \mathtt{nil} \wedge A$ where A is atomic.
2. $e\ \mathbf{q}\ Z(\mathbf{t}) \triangleq Z(\mathbf{t}) \wedge Z^*(\mathbf{t}, e)$ where Z is a predicate variable.
3. $e\ \mathbf{q}\ A \wedge B \triangleq (\mathtt{car}(e)\ \mathbf{q}\ A) \wedge (\mathtt{cdr}(e)\ \mathbf{q}\ B)$
4. $e\ \mathbf{q}\ A \vee B \triangleq (\mathtt{car}(e))\downarrow \wedge\ (\mathtt{car}(e) = \mathtt{true} \supset (\mathtt{cdr}(e)\ \mathbf{q}\ A))$
   $\wedge\ (\mathtt{car}(e) \neq \mathtt{true} \supset (\mathtt{cdr}(e)\ \mathbf{q}\ B))$
5. $e\ \mathbf{q}\ A \supset B \triangleq \forall r.\ A \wedge (r\ \mathbf{q}\ A) \supset (e(r)\ \mathbf{q}\ B)$
6. $e\ \mathbf{q}\ \forall x.A \triangleq \forall x.(e(x)\ \mathbf{q}\ A)$
7. $e\ \mathbf{q}\ \exists x.A \triangleq (\mathtt{cdr}(e)\ \mathbf{q}\ A_x[\mathtt{car}(e)]) \wedge A_x[\mathtt{car}(e)]$
8. $e\ \mathbf{q}\ (\lambda x.A)(t) \triangleq e\ \mathbf{q}\ A_x[t]$
9. $e\ \mathbf{q}\ (\mu X.\langle \lambda \mathbf{x}.A, \lambda \mathbf{xy}.B \rangle(\mathbf{t})) \triangleq$
   $(\mu X^*.\langle \lambda \mathbf{x}.A, \lambda \mathbf{x}r.(r\ \mathbf{q}\ A), \lambda \mathbf{xy}.B, \lambda \mathbf{xy}s.(s\ \mathbf{q}\ B); X, X^*, Y, Y^* \rangle)(\mathbf{t}, e)$
10. $e\ \mathbf{q}\ (\mu Y.\langle \lambda \mathbf{x}.A, \lambda \mathbf{xy}.B \rangle(\mathbf{t}, \mathbf{u})) \triangleq$
    $(\mu Y^*.\langle \lambda \mathbf{x}.A, \lambda \mathbf{x}r.(r\ \mathbf{q}\ A), \lambda \mathbf{xy}.B, \lambda \mathbf{xy}s.(s\ \mathbf{q}\ B); X, X^*, Y, Y^* \rangle)(\mathbf{t}, \mathbf{u}, e)$

All but clauses 2, 9, and 10 are quite standard. Let us explain these three definitions following Tatsuta.

Clause 2 is for a predicate variable which will be used through inductive definition. Clauses 9 and 10 represent the realizability of inductively defined predicates. The realizability interpretation for inductively defined predicates $\mu X.A$ is defined so that the realizer of $(\mu X.A)(t)$ and that of $A_X[\mu X.A](t)$ are the same. So we want to define $e\ \mathbf{q}\ (\mu X.A)(t)$ as $e\ \mathbf{q}\ A_X[\mu X.A](t)$. However, by expanding $e\ \mathbf{q}\ A_X[\mu X.A](t)$, we shall encounter $e\ \mathbf{q}\ (\mu X.A)(s)$, which we are defining now. By defining $X^*(t, e)$ as $e\ \mathbf{q}\ X(t)$, we have another inductive definition $X^*(t, e) \leftrightarrow (e\ \mathbf{q}\ A(t))$, which may contain X^* recursively. (This is a valid definition by the **HMONO-Q** condition, which ensures the monotonicity in this form.) This is the intuition of the clauses 9 and 10.

Remark

We defined the clauses 9 and 10 for the R-order only, and not for the general $R(n, df)$-order, since our realizability interpretation is for **BT+HMID**$_0$, and not for the full theory **BT+HMID**. We think our realizability interpretation may be extended to the general case in the same pattern as these clauses, but we have not done the details since the theory **BT+HMID**$_0$ is sufficient for our needs.

If the formula $(e \mathbf{\ q\ } A)$ is proved, we say "the formula A is provably realized by the term e". Now we can define the condition **HMONO-Q**.

Definition 27 (HMONO-Q) *The condition* **HMONO-Q**$(\lambda\mathbf{x}.A, \lambda\mathbf{xy}.B)$ *is defined as the quadruple of abstracts* $\langle \lambda\mathbf{x}.A, \lambda\mathbf{x}r.(r \mathbf{\ q\ } A), \lambda\mathbf{xy}.B, \lambda\mathbf{xy}s.(s \mathbf{\ q\ } B)\rangle$ *is monotone wrt. R4-order.*

We write $\mu X.\langle \lambda\mathbf{x}.A, \lambda\mathbf{x},\mathbf{y}.B\rangle$ and $\mu Y.\langle \lambda\mathbf{x}.A, \lambda\mathbf{x},\mathbf{y}.B\rangle$ as L_0 and L_1, and $\mu X.Q$, $\mu X^*.Q$, $\mu Y.Q$, and $\mu Y^*.Q$ as M_0, M_1, M_2 and M_3 where Q is $\langle \lambda\mathbf{x}.A, \lambda\mathbf{x}r.(r \mathbf{\ q\ } A), \lambda\mathbf{xy}.B, \lambda\mathbf{xy}s.(s \mathbf{\ q\ } B)\rangle$

Lemma 4 *Assuming the* **HMONO** *and* **HMONO-Q** *conditions hold, we have that* $L_0(\mathbf{x}) \leftrightarrow M_0(\mathbf{x})$ *and* $L_1(\mathbf{x},\mathbf{y}) \leftrightarrow M_2(\mathbf{x},\mathbf{y})$

Proof. We easily have $\langle L_0, L_1\rangle \leq_R \langle M_0, M_2\rangle$. We shall prove the opposite direction. We have $\langle L_0, M_1, L_1, M_3\rangle \leq_{R4} \langle M_0, M_1, M_2, M_3\rangle$. Applying these elements to $\langle \lambda\mathbf{x}.A, \lambda\mathbf{x}r.(r \mathbf{\ q\ } A), \lambda\mathbf{xy}.B, \lambda\mathbf{xy}s.(s \mathbf{\ q\ } B)\rangle$, we have

$$\langle L_0, \lambda\mathbf{x}r.(r \mathbf{\ q\ } A)\theta, L_1, \lambda\mathbf{xy}s.(s \mathbf{\ q\ } B)\theta\rangle \leq_{R4} \langle M_0, M_1, M_2, M_3\rangle$$

where θ is $\{X := L_0, X^* := M_1, Y := L_1, Y^* := M_3\}$. By the definition of $\leq_{R4}$ and $\langle L_0, L_1\rangle \leq_R \langle M_0, M_2\rangle$, we can replace the right-hand-side by $\langle L_0, M_1, L_1, M_3\rangle$, and get

$$\langle L_0, \lambda\mathbf{x}r.(r \mathbf{\ q\ } A)\theta, L_1, \lambda\mathbf{xy}s.(s \mathbf{\ q\ } B)\theta\rangle \leq_{R4} \langle L_0, M_1, L_2, M_3\rangle$$

Taking this formula as an assumption of the μ-ind-1 rule (besides the **HMONO** and **HMONO-Q** conditions), we obtain $\langle M_0, M_1, M_2, M_3\rangle \leq_{R4} \langle L_0, M_1, L_1, M_3\rangle$ by μ-ind-1. This formula implies $\langle M_0, M_2\rangle \leq_R \langle L_0, L_1\rangle$.

4.4.2 Soundness of Realizability Interpretation

We will prove the following soundness theorem.

Theorem 15 (Soundness of q-realizability for BT+HMID$_0$)
Let $A_1 \cdots, A_n, F$ *be formulas. If*

$$A_1 \cdots, A_n \vdash_{\mathbf{BT+HMID}_0} F$$

holds, then we have

$$A_1, \cdots, A_n, (x_1 \mathbf{\ q\ } A_1), \cdots, (x_n \mathbf{\ q\ } A_n) \vdash_{\mathbf{BT+HMID}} r \mathbf{\ q\ } F$$

for some term r.

Since the only essential difference between Tatsuta's theory and our theory is **HMID**$_0$, we only have to consider **HMID**$_0$ to prove this theorem.

Lemma 5 *Suppose that the following conditions hold for* A *and* B.

$$\mathbf{HMONO}(A, B)$$
$$\mathbf{HMONO\text{-}Q}(A, B)$$

Then, the μ*-eq-1 rule is realized.*

Proof. By the induction on the abstracts A and B, we can prove that the same term is a realizer of both sides of the μ-eq-1 rule.

Lemma 6 *The* μ*-ind-1 rule is realized.*

Proof. Suppose that the following conditions hold for A and B.

$$\mathbf{HMONO}(A, B)$$
$$m \mathbf{\ q\ } \mathbf{HMONO}(A, B)$$
$$\mathbf{HMONO\text{-}Q}(A, B)$$

We also assume the following for an n-ary abstract F.

$$\langle A[F, M_Y], B[F, M_Y]\rangle \leq_R \langle F, M_Y\rangle$$
$$a \mathbf{\ q\ } \langle A[F, M_Y], B[F, M_Y]\rangle \leq_R \langle F, M_Y\rangle$$

By μ`-ind-1`, the first one implies $M_X \leq_N F$, which in turn implies $M_X \leq_N A[F, M_Y]$.

The second one implies

$$\forall\mathbf{x}r.(A[F, M_Y] \wedge (r \mathbf{\ q\ } A[F, M_Y]) \supset (\mathtt{car}(a)\mathbf{x}r \mathbf{\ q\ } F(\mathbf{x}))) \tag{4.1}$$

Since we assume **HMONO-Q**, the quadruple $\langle \lambda\mathbf{x}.A, \lambda\mathbf{x}r.r \mathbf{\ q\ } A, \lambda\mathbf{xy}.B, \lambda\mathbf{xy}s.s \mathbf{\ q\ } B\rangle$ has a least fixpoint. We know that the first and the third elements of the least fixpoint are M_X and M_Y by Lemma 4. We abbreviate the second and the fourth elements as L_X and L_Y.

Our goal is to show the following formula:

$$\forall\mathbf{x}r.\ M_X(\mathbf{x}) \wedge L_X(\mathbf{x}, r) \supset (f\mathbf{x}r \mathbf{\ q\ } F(\mathbf{x})))$$

for an appropriate term f.

Let H be $\lambda\mathbf{x}r.(L_X(\mathbf{x}, r) \wedge (M_X(\mathbf{x}) \supset (f\mathbf{x}r \mathbf{\ q\ } F(\mathbf{x}))))$. Applying the quadruple $\langle M_X, H, M_Y, L_Y\rangle$ to the μ-ind-0 rule (in the case of R4-order) with a little calculation, it becomes:

$$\frac{\lambda \mathbf{x}r.(r \ \mathbf{q} \ A[X, M_Y])\theta \leq_N H}{L_X \leq_N H}$$

where θ is $\{X := M_X, X^* := H\}$. Then all we have to prove is

$$\forall \mathbf{x}, r.\ (r \ \mathbf{q} \ A[X, M_Y])\theta \supset H(\mathbf{x}, r)$$

for an appropriate term f. This is equivalent to conjunction of the following formulas:

$$\forall \mathbf{x}, r.\ (r \ \mathbf{q} \ A[X, M_Y])\theta \supset L_X(\mathbf{x}, r) \tag{4.2}$$

$$\forall \mathbf{x}, r.\ (r \ \mathbf{q} \ A[X, M_Y])\theta \supset M_X(\mathbf{x}) \supset (f\mathbf{x}r \ \mathbf{q} \ F(\mathbf{x})) \tag{4.3}$$

By **HMONO-Q** condition, we have the simple monotonicity for $(r \ \mathbf{q} \ A[X, Y])$ in X^*. Hence, we can replace H by L_X in θ, and we obtain the goal (4.2).

In the following, we will prove the goal (4.3). We fix $\mathbf{x}$ and r, and assume $(r \ \mathbf{q} \ A[X, M_Y])\theta$.

By expanding $m \ \mathbf{q} \ \mathbf{HMONO}(A, B)$, and putting

$$\begin{aligned}
X_0 &\triangleq M_X \\
X_0^* &\triangleq H \\
X_1 &\triangleq F \\
X_1^* &\triangleq \lambda \mathbf{x}r.(M_X(\mathbf{x}) \wedge L_X(\mathbf{x}, r) \wedge (r \ \mathbf{q} \ F(\mathbf{x}))) \\
Y_i &\triangleq M_Y \quad (i = 0, 1) \\
Y_i^* &\triangleq L_Y \quad (i = 0, 1)
\end{aligned}$$

we get $M_X(\mathbf{x}) \wedge L_X(\mathbf{x}, r) \supset (t \ \mathbf{q} \ A)\phi$ where

$$\phi \triangleq \{X := F, X^* := \lambda \mathbf{x}r.\ (M_X(\mathbf{x}) \wedge L_X(\mathbf{x}, r) \wedge (r \ \mathbf{q} \ F(\mathbf{x}))), Y := M_Y, Y^* := L_Y\}$$

and

$$t \triangleq (\mathtt{car}(m(\langle f, \langle \lambda \mathbf{x}\mathbf{y}s.s, \lambda \mathbf{x}\mathbf{y}rs.s\rangle\rangle))\mathbf{x}r$$

By the simple monotonicity in X_0^*, we may replace ϕ by

$$\psi \triangleq \{X := F, X^* := \lambda \mathbf{x}r.(r \ \mathbf{q} \ F(\mathbf{x})), Y := M_Y, Y^* := L_Y\}$$

Note that $(t \ \mathbf{q} \ A)\psi$ is equivalent to $(t \ \mathbf{q} \ A[F, M_Y])$. From (4.1) and this formula, we have $M_X(\mathbf{x}) \wedge L_X(\mathbf{x}, r) \supset \mathtt{car}(a)\mathbf{x}t \ \mathbf{q} \ F(\mathbf{x})$. We already have (2), so we can delete $L_X(\mathbf{x}, r)$ from this formula. By the fixpoint theorem, we can take f as a fixpoint of the equation $f\mathbf{x}r = \mathtt{car}(a)\mathbf{x}t$. Then we have $M(\mathbf{x}) \supset (f\mathbf{x}r \ \mathbf{q} \ F(\mathbf{x}))$ which is the conclusion of the goal (4.3).

By these lemmas, we have the soundness theorem. We also have the following corollaries.

Corollary 1 (Disjunction Property and Term-Existence Property) *1. Disjunction Property holds in* $\mathbf{BT{+}HMID_0}$*; namely, from a proof of the formula* $A \vee B$*, we can decide which of* A *or* B *holds.*

2. Term Existence Property holds in $\mathbf{BT{+}HMID_0}$*; namely, from a proof of the formula* $\exists x.\ A(x)$*, we can effectively find a term* t *such that* $A(t)$ *holds.*

Corollary 2 (Program Extraction Theorem) *From a proof of the formula* $\forall x.\exists y.\ A(x, y)$ *in* $\mathbf{BT{+}HMID_0}$*, we can effectively find a term* f *and the proof of* $\forall x.\ A(x, f(x))$ *in* $\mathbf{BT{+}HMID}$.

By Program Extraction Theorem, we can derive programs in our theory **BT+HMID**, in the style of constructive programming.

4.4.3 Verification of the HMONO-Q condition

In this subsection, we verify the **HMONO-Q** condition for the prime number example in Section 4.3.6.

Let A and B be the same as in the prime number example.

Suppose $\langle X_0, X_0^*, Y_0, Y_0^*\rangle \leq_{R4} \langle X_1, X_1^*, Y_1, Y_1^*\rangle$. Then, our goal is to prove the following formula:

$$\begin{aligned}
&\langle A, \lambda nr.(r \ \mathbf{q} \ A(n)), B, \lambda ns.(s \ \mathbf{q} \ B(n))\rangle\theta_0 \\
&\quad \leq_{R4} \langle A, \lambda nr.(r \ \mathbf{q} \ A(n)), B, \lambda ns.(s \ \mathbf{q} \ B(n))\rangle\theta_1
\end{aligned}$$

where θ_i is $\{X := X_i, X^* := X_i^*, Y := Y_i, Y^* := Y_i^*\}$.

Since X appears positively in A, it is easily seen that $A\theta_0 \leq_N A\theta_1$ and $\lambda nr.(r \ \mathbf{q} \ A(n))\theta_0 \leq_N \lambda nr.(r \ \mathbf{q} \ A(n))\theta_1$ hold. We also have $\langle A, B\rangle\theta_0 \leq_R \langle A, B\rangle\theta_1$, which is the **HMONO** condition.

The remaining goal is to prove the following:

$$\langle B, \lambda ns.(s \ \mathbf{q} \ B(n))\rangle\theta_0 \leq_R \langle B, \lambda ns.(s \ \mathbf{q} \ B(n))\rangle\theta_1$$

This is conjunction of three formulas, but here we shall prove the following one only.

$$\lambda ns.(s \ \mathbf{q} \ B(n))\theta_0 \leq_N \lambda ns.(s \ \mathbf{q} \ B(n))\theta_1$$

The formula $(s \ \mathbf{q} \ B(n))$ is expanded to the following:

$$\begin{aligned}
&X(n) \wedge X^*(n, \mathtt{car}(s)) \wedge (\mathtt{cadr}(s) \ \mathbf{q} \ (1 < x)) \\
&\quad \wedge \forall m, r.\ ((m < n) \wedge (r \ \mathbf{q} \ m < n) \supset \\
&\qquad\qquad X(m) \wedge X^*(m, \mathtt{car}(\mathtt{cddr}(s)mr)) \\
&\qquad\qquad \wedge \forall u.(Y(m) \wedge Y^*(m, u) \supset (\mathtt{cdr}(\mathtt{cddr}(s)mr)u \ \mathbf{q} \ \neg\mathtt{Div}(m, n))))
\end{aligned}$$

Since the only occurrence of Y^* appears in the subformula $Y(m) \wedge Y^*(m, u)$ and $Y_0(z)$ implies $\forall u.Y_0^*(z, u) \leftrightarrow Y_1^*(z, u)$, we have $\lambda ns.(s \ \mathbf{q} \ B(n))\theta_0 \leq_N \lambda ns.(s \ \mathbf{q} \ B(n))\theta_2$ where θ_2 is $\{X := X_0, X^* := X_0^*, Y := Y_0, Y^* := Y_1^*\}$. We also have $X_0(z)$ implies $Y_0(z) \leftrightarrow Y_1(z)$, so we have $\lambda ns.(s \ \mathbf{q} \ B(n))\theta_2 \leq_N \lambda ns.(s \ \mathbf{q} \ B(n))\theta_3$ where θ_3 is $\{X := X_0, X^* := X_0^*, Y := Y_1, Y^* := Y_1^*\}$. The occurrences of X and X^* are positive, and we finally have $\lambda ns.(s \ \mathbf{q} \ B(n))\theta_3 \leq_N \lambda ns.(s \ \mathbf{q} \ B(n))\theta_1$. Combining these results, we get $\lambda ns.(s \ \mathbf{q} \ B(n))\theta_0 \leq_N \lambda ns.(s \ \mathbf{q} \ B(n))\theta_1$, which is the conclusion.

Remark

This proof did not use the components of A and B in detail; it only mentioned the occurrences of X and Y. That the **HMONO**(A, B) and **HMONO-Q**(A, B) conditions hold come from the following facts:

- X appears positively in A and B
- Y may appear negatively in B, but the only negative occurrence of Y has a preceding occurrence of X.

The second fact can be made more precise: If every occurrence of Y is in a subformula of the form of $X(x) \wedge (Y(x, y) \supset \cdots)$ where x is the overlapping argument, then we say Y has a preceding occurrence of X. (In the above example, we did not have an extra argument y in Y, but it is easily seen that the proof will not be affected by existence of such an argument.) We can see these two facts are keys to prove the two conditions, and whenever these two facts hold, we have the **HMONO** and **HMONO-Q** conditions.

One may think the verification of the **HMONO-Q** condition is too complex and hence our induction mechanism is hard to use. However, all the examples we have considered fall in this uniform pattern, and we believe its verification is not so problematic.

4.5 Provability Relations and its Refinement

In this section, we define a basic version of provability relation and its variants using **HMID**$_0$ mechanism.

4.5.1 Defining a Provability Relation

We shall simultaneously define a unary predicate `Prop` and a binary predicate `Proves` which formalizes the following two concepts.

$$\begin{array}{c}\texttt{Prop}(p) \text{ represents ``}p\text{ is a proposition''} \\ \texttt{Proves}(a,p) \text{ represents ``}a\text{ is a proof of }p\text{''}\end{array}$$

In ordinary predicate logic, the first concept is defined by itself, and does not depend on the second one. However, we are mainly interested in systems like Frege structures, Martin-Löf's Type Theory, or Sato's $\mathcal{RPT}$(Reflective Proof Theory[34]). One of the characteristic points of these theories is that the above two concepts essentially refers to each other, so we cannot define the first concept independently.

We will henceforth assume that all logical connectives are encoded by appropriate terms using corresponding constants, for instance, $a\dot{\wedge}b$ is $\langle \mathbf{c}_\wedge\, a\, b\rangle$ where $\mathbf{c}_\wedge$ is a constant uniquely associated with $\wedge$.

$$\begin{array}{lll}
A & \triangleq \lambda x. & x = \dot{\bot} \\
& & \vee\ \exists a.\exists b.\ x = a\dot{=}b \\
& & \vee\ \exists a.\ x = \dot{N}(a) \\
& & \vee\ \exists y.\exists z.\ x = (y\dot{\wedge}z) \wedge X(y) \wedge X(z) \\
& & \vee\ \exists y.\exists z.\ x = (y\dot{\vee}z) \wedge X(y) \wedge X(z) \\
& & \vee\ \exists y.\exists z.\ x = (y\dot{\supset}z) \wedge X(y) \wedge \forall w.(Y(w, y) \supset X(z)) \\
& & \vee\ \exists y.\ x = (\dot{\forall}y) \wedge \forall a.X(y(a)) \\
& & \vee\ \exists y.\ x = (\dot{\exists}y) \wedge \forall a.X(y(a))
\end{array}$$

$$\begin{array}{lll}
B & \triangleq \lambda rx. & X(x) \wedge \\
& & (\exists a.\exists b.\ x = (a\dot{=}b) \wedge a = b \wedge r = \texttt{nil} \\
& & \vee\ \exists a.\ x = \dot{N}(a) \wedge \texttt{Nat}(a) \wedge r = \texttt{nil} \\
& & \vee\ \exists y.\exists z.\exists a.\exists b.\ x = (y\dot{\wedge}z) \wedge r = \langle a, b\rangle \wedge Y(a, y) \wedge Y(b, z) \\
& & \vee\ \exists y.\exists z.\exists a.\exists b.\ x = (y\dot{\vee}z) \wedge r = \langle a, b\rangle \wedge a\downarrow \\
& & \qquad\quad \wedge\ (a = \texttt{true} \supset Y(b, y)) \wedge (a \neq \texttt{true} \supset Y(b, z)) \\
& & \vee\ \exists y.\exists z.\ x = (y\dot{\supset}z) \wedge X(y) \wedge \texttt{fun?}(r) = \texttt{true} \\
& & \qquad\quad \wedge\ \forall w.(Y(w, y) \supset Y(r(w), z)) \\
& & \vee\ \exists y.\ x = \dot{\forall}y \wedge \forall a.\ Y(r(a), y(a)) \\
& & \vee\ \exists y.\exists a.\exists b.\ x = \dot{\exists}y \wedge r = \langle a, b\rangle \wedge Y(a, y(b)))
\end{array}$$

Here, `Nat` is the predicate defined before. Quantified formulas such as $\forall x.p(x)$ and $\exists x.p(x)$ are represented by $\dot{\forall}(\lambda x.p)$ and $\dot{\exists}(\lambda x.p)$, respectively. We can add other kinds of propositions such as $(a\downarrow)$. We did not do so simply because we want to show the essential feature of the mechanism.

Both templates contain negative occurrences of Y. **BT+HMID**$_0$ allows such occurrences provided **HMONO**$(A, B; X, Y)$ and **HMONO-Q**$(A, B; X, Y)$ are satisfied. Note that, in the above formulation, we bravely changed the order of the overlapping argument x and the non-overlapping argument r in Y. Strictly speaking, x must come first, but it can be adjusted easily.

Lemma 7 *For the above A and B,* **HMONO**$(A, B; X, Y)$ **HMONO-Q**$(A, B; X, Y)$ *hold.*

This lemma is proved in just the same as the case for Prime Numbers. The only negative occurrence of Y in A is in the subformula of the form $X(y) \wedge \forall w.(Y(w, y) \supset \cdots)$, so $A_X[X_0, Y_0]$ is equivalent to $A_X[X_0, Y_1]$ provided that $\langle X_0, Y_0\rangle \leq_R \langle X_1, Y_1\rangle$. From this observation, we have $A_X[X_0, Y_0] \leq_N A_X[X_1, Y_1]$. Similarly, we have other cases and reach the conclusion.

From this lemma, we can safely define $\langle \texttt{Prop}, \texttt{Proves}\rangle$ by the $\mathbf{HMID}_0$ mechanism.

Definition 28

$$\begin{aligned}
\texttt{Prop} &\triangleq \mu X.\ \langle A, B; X, Y\rangle \\
\texttt{Proves} &\triangleq \mu Y.\ \langle A, B; X, Y\rangle \\
\texttt{True} &\triangleq \lambda x.\exists r.\ \texttt{Proves}(r, x)
\end{aligned}$$

The pair $\langle \texttt{Prop}, \texttt{Proves}\rangle$ is a provability relation.

We can infer that Prop is the least fixpoint of A, while we cannot infer Proves is that of B, by the restriction for the μ-ind-1 rule. If we would omit the first conjunct $X(x)$ in the definition of B, the following (intuitively true) fact would not be provable in $\mathbf{BT}+\mathbf{HMID}_0$.

$$\forall x.(\texttt{True}(x) \supset \texttt{Prop}(x))$$

However, we need induction only for Prop in practice, so μ-ind-1 is sufficient for all our needs.

4.5.2 Re-defining the Provability Relation

It is well known that Harrop formulas do not carry computational meaning. Hence, we may attach as a proof-term a dummy constant nil to Harrop formulas. This optimization technique is quite useful as is shown in (Ref. [21]). Here we define a refined relation and prove equivalence of the original one and this version.

$$\begin{aligned}
H \triangleq \lambda x.\quad & x = \dot{\bot} \\
& \vee\ \exists a.\exists b.\ x = (a \dot{=} b) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\wedge} z) \wedge X(y) \wedge X(z) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\supset} z) \wedge \texttt{Prop}(y) \wedge X(z) \\
& \vee\ \exists y.\ x = (\dot{\forall} y)) \wedge \forall a.X(y(a))
\end{aligned}$$

This is a usual positive inductive definition, so $\mathbf{HMONO}(H; X)$ and $\mathbf{HMONO\text{-}Q}(H; X)$ are clearly satisfied. Let Harrop be $\mu X.H$.

We can prove the following important properties of Harrop formulas.

Theorem 16
1. $\texttt{Harrop}(x) \supset \texttt{Prop}(x)$
2. $\exists f.\forall x.(\texttt{Harrop}(x) \supset \texttt{True}(x) \supset \texttt{Proves}(f, x))$

These are proved by the induction on Harrop.

We then define the following new template B_1.

$$B_1 \triangleq \lambda rx.\ B(x, r) \vee \texttt{Harrop}(x) \wedge \texttt{True}(x) \wedge r = \texttt{nil}$$

Using $\langle A, B_1\rangle$, we define a new provability relation.

Definition 29

$$\begin{aligned}
\texttt{Prop}_1 &\triangleq \mu X.\langle A, B_1; X, Y\rangle \\
\texttt{Proves}_1 &\triangleq \mu Y.\langle A, B_1; X, Y\rangle \\
\texttt{True}_1 &\triangleq \lambda x.\exists r.\texttt{Proves}_1(r, x)
\end{aligned}$$

The pair $\langle A, B_1\rangle$ also satisfies **HMONO** and **HMONO-Q**, so we can make use of μ-eq-1 and μ-ind-1.

We have the following theorem, which shows equivalence of two provability relations.

Theorem 17 *We have that*

$$\forall x.\ \texttt{Prop}(x) \leftrightarrow \texttt{Prop}_1(x),$$

and

$$\begin{aligned}
\forall x.\ (\texttt{Prop}(x) \supset\ & \exists f.\forall r.\ (\texttt{Proves}(r, x) \supset \texttt{Proves}_1(f(r), x)) \\
& \wedge \exists g.\forall r.\ (\texttt{Proves}_1(r, x) \supset \texttt{Proves}(g(r), x)))
\end{aligned}$$

Proof. (Sketch) Let $\texttt{IH}(x)$ (standing for induction hypothesis) be the following formula:

$$\begin{aligned}
& \texttt{Prop}(x) \wedge \texttt{Prop}_1(x) \\
& \wedge\ \exists f.\forall r.(\texttt{Proves}(r, x) \supset \texttt{Proves}_1(f(r), x)) \\
& \wedge\ \exists g.\forall r.(\texttt{Proves}_1(r, x) \supset \texttt{Proves}(g(r), x))
\end{aligned}$$

We can prove $\forall x.\texttt{Prop}(x) \supset \texttt{IH}(x)$ and $\forall x.\texttt{Prop}_1(x) \supset \texttt{IH}(x)$ by the induction on Prop and $\texttt{Prop}_1$. Using Theorem 16, we can always create a candidate proof for a Harrop formula, hence we can calculate g from the proof of $\texttt{Harrop}(x)$.

We can improve the code further; for instance, we can optimize a program corresponding to a disjunctive formula if it is decidable. For example, let us see the following formula.

$$\lambda x.\ (x = 0 \wedge A(x)) \vee \exists yz.(x = \langle y, z\rangle \wedge B(x))$$

Suppose A and B are Harrop propositions. This formula contains disjunction and the existential quantifier, so it is not Harrop by definition. However, its proof carries no more information than x. We can, therefore, add this kind of propositions to Harrop propositions, and successfully reduce the program (proof).

Moreover, we can proceed to optimization of arbitrary self realizing formulas[3]. If we know a formula is self-realizing (which is a *semantic* notion, on the contrary to Harrop formula, which is a *syntactic* notion in ordinary logic.), then a realizer for this formula is redundant and we can eliminate it. This kind of optimization seems to have practical use, yet we have not studied this topic in detail.

In constructive programming, we often encounter inefficient programs. As is shown in the previous subsections, we can define several kinds of `(Prop, Proves)` in the theory $\mathbf{BT{+}HMID_0}$, and can prove equivalence of them. This corresponds to re-defining a new realizability interpretation, and proving its soundness, so the results of optimization may be the same. However, the point here is that we can prove meta-theorems such as equivalence of two definitions *within* the theory. Note that, we can further extract an optimization program from the meta-theorems by the program extraction theorem, which follows from the realizability interpretation. This point is one of major good points in our theory. For example:

Corollary 3 *In the previous theorem, we can effectively find functions f and g from the proof.*

This is an immediate consequence of the previous theorem together with the program extraction theorem.

Extracting programs from proofs as well as meta-proofs seems a quite promising paradigm in constructive programming. In this chapter, we just presented foundation of a candidate theory, realizability interpretation, and its relation to other theories. However, we may be able to do *reflective constructive programming* in future based on the technique presented in this chapter.

4.6 Interpretation of Other Theories

In this section, we show how other theories can be interpreted by the $\mathbf{HMID_0}$ mechanism. We first interpret the Logical Theory of Constructions, and then we briefly mention how to interpret Martin-Löf's type theory.

4.6.1 The Logical Theory of Constructions

The Logical Theory of Constructions (**LTC**)[3] is a special logic which has Aczel's Frege structures as its origin, and is used to interpret Martin-Löf's Type Theory **ITT**. We may say **LTC** is a formalized version of (a generalization of) Frege structures.

LTC is actually a family of theories, $\mathbf{LTC}_0, \mathbf{LTC}_1, \cdots, \mathbf{LTC}_\omega$. $\mathbf{LTC}_0$ is the basic theory, and $\mathbf{LTC}_{i+1}$ is a reflected version of $\mathbf{LTC}_i$. $\mathbf{LTC}_\omega$ is the union of each finite level of reflection.

[3]A self realizing formula is a formula $A(x)$, for which there exists a term f, the formula $\forall x.\ f(x)\ \mathbf{q}\ A$ holds.

Terms of **LTC** is essentially combinatory terms with primitive function symbols for natural numbers, lambda terms, and others. Some of them such as 0, $Succ$, λ, $\langle \_, \_ \rangle$, Inl, Inr are canonical. There are non-canonical function symbols corresponding to these canonical symbols. There are constants which internally represent logical connectives such as $\dot{\wedge}, \dot{\supset}$ and so on. Atomic formulas are $\bot$, $a = b$, $a \leadsto b$, $a \Rightarrow b$. For a typographic reason, we use a different symbol $\Rightarrow$ from the original paper[3]. In $\mathbf{LTC}_i$ for $i \geq 1$, we have two more atomic formulas $T(a)$ and $P_j(a)$ (for $j \leq i$).

We briefly summarize intuitive meaning of atomic formulas. The formula $a = b$ means a and b are *definitionally* equal. The evaluation mechanism is call-by-name, and it is captured by $\leadsto$. The formula $a \leadsto b$ means that a term a evaluates to a canonical form b. Since this relation $\leadsto$ evaluates a term into a canonical form only (a canonical term is a term whose outermost function symbol, and it possibly contains redexes), we have another relation $\Rightarrow$ which represents "full evaluation". Two atomic formulas are important for reflection: "a term a is a proposition" (denoted as $P_i(a)$) and "a term a is a true proposition"(denoted as $T(a)$). Since reflection is repeated at finitely many times, $P_i(a)$ has a suffix i which represents the level of reflection. Besides them, there are ordinary first-order logical connectives as well as quantifiers for functions $\forall^1$ and $\exists^1$. The predicate for natural numbers $Nat(a)$ can be defined as $\exists n.(a \Rightarrow n)$.

The logic of $\mathbf{LTC}_0$ is an ordinary intuitionistic predicate calculus with equality.

Extending $\mathbf{LTC}_0$ by reflection, we get $\mathbf{LTC}_1$. The rules (called reflection rules) for $P_1(a)$ and $T(a)$ are as follows. These rules give connection of internal logic (a in $T(a)$) to external logic (logic of $\mathbf{LTC}_0$).

$$\frac{\Phi^1_c}{P_1(c)} \qquad \frac{\Phi^1_c}{T(c) \leftrightarrow \Psi_c}$$

where Φ^1_c and Ψ_c are defined by the following table.

c	Φ^1_c	Ψ_c
$(a \dot{=} b)$	T	$(a = b)$
$(a \dot{\leadsto} b)$	T	$(a \leadsto b)$
$(a \dot{\Rightarrow} b)$	T	$(a \Rightarrow b)$
$\dot{\bot}$	T	$\bot$
$(a \dot{\wedge} b)$	$P_1(a) \wedge P_1(b)$	$T(a) \wedge T(b)$
$(a \dot{\vee} b)$	$P_1(a) \wedge P_1(b)$	$T(a) \vee T(b)$
$(a \dot{\supset} b)$	$P_1(a) \wedge (T(a) \supset P_1(b))$	$T(a) \supset T(b)$
$(\dot{\forall}(p))$	$\forall x.P_1(p(x))$	$\forall x.T(p(x))$
$(\dot{\exists}(p))$	$\forall x.P_1(p(x))$	$\exists x.T(p(x))$
$(\dot{\forall}^1(p))$	$\forall^1 f.P_1(p(f))$	$\forall^1 f.T(p(f))$
$(\dot{\exists}^1(p))$	$\forall^1 f.P_1(p(f))$	$\exists^1 f.T(p(f))$

Next, we give rules for $\mathbf{LTC}_i$ for $i > 1$. In the following, Φ^i_c and Ψ^i_c is in a row of the above table or the following new table.

$$\frac{\Phi^i_c}{P_i(c)} \qquad \frac{\Phi^i_c}{T(c) \leftrightarrow \Psi_c} \qquad \frac{P_j(c)}{P_i(c)}\,(j < i)$$

c	Φ^i_c	Ψ_c
$\dot{P}_j(a)$ $(with j < i)$	T	$P_j(a)$

LTC$_i$ (for $i \geq 0$) is the theory defined as above. We therefore have an increasing hierarchy of theories **LTC**$_0$, **LTC**$_1$, $\cdots$. As a limit, we have **LTC**$_\omega$, the union of **LTC**$_i$ for $i \geq 0$. Note that we have no induction rules.

4.6.2 Interpreting LTC by BT+HMID$_0$

After defining the provability relation in the previous section, it is now straightforward to interpret **LTC**$_0$ by **BT+HMID**$_0$.

Variables in **LTC**$_0$ is uniquely translated into variables in **BT+HMID**$_0$. All the terms in **LTC**$_0$ is injectively interpreted as some terms in **BT+HMID**$_0$. For the sake of simplicity, we assume that terms like $\dot{P}_i(a)$ and $\dot{T}(a)$ are already contained in the language of **LTC**$_0$ (with no rules about reflection).

To interpret the notion of terms, two predicates `Term` and `Func` are inductively defined; `Term`(a) means a is an interpretation of a term, and `Func`(a) means a is an interpretation of a function. Two more predicates are inductive defined; `Eval` and `FullEval`. They are used to interpret $\rightsquigarrow$ and $\Rightarrow$. We do not give their detailed definitions here.

Every formula in **LTC**$_0$ is translated into a term in **BT+HMID**$_0$ preserving the structure and the set of free variables. For example, $\exists y.\ x = Succ(y)$ is translated into $\dot{\exists}(\lambda y.\ (x \dot{=} SUCC(y)))$ where $Succ(\_)$ is translated into $SUCC(\_)$.

We now define two templates.

$$\begin{aligned}
A_0 \;\triangleq\; \lambda x.\quad & x = \dot{\bot} \\
& \vee\ \exists a.\exists b.\ x = (a \dot{=} b) \\
& \vee\ \exists a.\exists b.\ x = (a \dot{\rightsquigarrow} b) \\
& \vee\ \exists a.\exists b.\ x = (a \dot{\Rightarrow} b) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\wedge} z) \wedge X(y) \wedge X(z) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\vee} z) \wedge X(y) \wedge X(z) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\supset} z) \wedge X(y) \wedge (Y(y) \supset X(z)) \\
& \vee\ \exists y.\ x = (\dot{\forall} y) \wedge \forall a.(\mathtt{Term}(a) \supset X(y(a))) \\
& \vee\ \exists y.\ x = (\dot{\exists} y) \wedge \forall a.(\mathtt{Term}(a) \supset X(y(a))) \\
& \vee\ \exists y.\ x = (\dot{\forall}^1 y) \wedge \forall f.(\mathtt{Func}(f) \supset X(y(f))) \\
& \vee\ \exists y.\ x = (\dot{\exists}^1 y) \wedge \forall f.(\mathtt{Func}(f) \supset X(y(f)))
\end{aligned}$$

$$\begin{aligned}
B_0 \;\triangleq\; \lambda x.\quad & X(x) \wedge \\
& (\exists a.\exists b.\ x = (a \dot{=} b) \wedge (a = b) \\
& \vee\ \exists a.\exists b.\ x = (a \dot{\rightsquigarrow} b) \wedge \mathtt{Eval}(a, b) \\
& \vee\ \exists a.\exists b.\ x = (a \dot{\Rightarrow} b) \wedge \mathtt{FullEval}(a, b) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\wedge} z) \wedge Y(y) \wedge Y(z) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\vee} z) \wedge Y(y) \vee Y(z) \\
& \vee\ \exists y.\exists z.\ x = (y \dot{\supset} z) \wedge X(y) \wedge (Y(y) \supset Y(z)) \\
& \vee\ \exists y.\ x = (\dot{\forall} y) \wedge \forall a.(\mathtt{Term}(a) \supset Y(y(a))) \\
& \vee\ \exists y.\ x = (\dot{\exists} y) \wedge \exists a.(\mathtt{Term}(a) \wedge Y(y(a))) \\
& \vee\ \exists y.\ x = (\dot{\forall}^1 y) \wedge \forall f.(\mathtt{Func}(f) \supset Y(y(f))) \\
& \vee\ \exists y.\ x = (\dot{\exists}^1 y) \wedge \exists f.(\mathtt{Func}(f) \wedge Y(y(f)))
\end{aligned}$$

We take the least fixpoint as $\mu X.\langle A_0, B_0; X, Y\rangle$ and $\mu Y.\langle A_0, B_0; X, Y\rangle$, and call them `Prop`$_0$ and `True`$_0$. We omit the verification of **HMONO** and **HMONO-Q** conditions since it is almost the same as before.

Theorem 18 (Soundness of Translation) *Let F be a formula in* **LTC**$_0$. *Then,* `Prop`$_0(F')$ *is provable in* **BT+HMID**$_0$, *and if F is provable in* **LTC**$_0$, *then* `True`$_0(F')$ *is provable in* **BT+HMID**$_0$ *where F' is the translation of F.*

Each rule in **LTC**$_0$ is easily justified. Since **LTC**$_0$ does not have any induction rules, the restriction of the μ-ind-1 rule is not a problem.

We then go to the level 1. We assumed that the language of **LTC**$_0$ already contained terms like $\dot{P}_i(a)$, so the languages of **LTC**$_i$ are the same, and we continue to use the same definition of `Term` and so on.

We define the templates A_1 and B_1 as follows:

$$\begin{aligned}
A_1 \;\triangleq\;\ & \lambda x.\ A_0(x) \\
& \vee\ \exists a.\ \mathtt{Term}(a)\ \wedge\ x = \dot{P}_1(a) \\
& \vee\ \exists a.\ \mathtt{Term}(a)\ \wedge\ x = \dot{T}(a) \\
B_1 \;\triangleq\;\ & \lambda x.\ B_0(x) \\
& \vee\ \exists a.\ \mathtt{Term}(a)\ \wedge\ x = \dot{P}_1(a)\ \wedge\ \mathtt{Prop}_0(a) \\
& \vee\ \exists a.\ \mathtt{Term}(a)\ \wedge\ x = \dot{T}(a)\ \wedge\ \mathtt{True}_0(a)
\end{aligned}$$

We define `Prop`$_1 \triangleq \mu X.\langle A_1, B_1; X, Y\rangle$ and `True`$_0 \triangleq \mu Y.\langle A_1, B_1; X, Y\rangle$. Again, we omit the verification of **HMONO** and **HMONO-Q** conditions.

By repeating this kind of construction, we have the translation of **LTC**$_i$ for each $i \geq 0$, and **LTC**$_\omega$.

Theorem 19 (Soundness of Translation) *Let F be a formula in* **LTC**$_i$ *for $i \geq 0$. Then,* $\texttt{Prop}_i(F')$ *is provable in* **BT+HMID**$_0$, *and if F is provable in* **LTC**$_i$, *then* $\texttt{True}_i(F')$ *is provable in* **BT+HMID**$_0$ *where F' is the i-th level translation of F.*

4.6.3 Reformulating HMID$_0$ on LTC$_0$

Reflection in logic is internalization of a metatheory of the logic itself. In this respect, our theory is not really reflective, since our metatheory is **BT+HMID**$_0$, while our target logic is **LTC**-like logic (where an implicational formula $a \supset b$ may sometimes be a formula even if b is not a formula) up to now.

In order to have the same kind of logic as internal and external ones, we may adopt **LTC**$_0$ as our meta-theory, namely we may consider something like **LTC**$_0$ + **HMID**. It seems quite straightforward to move the **HMID** mechanism onto **LTC**$_0$ The resulting theory **LTC**$_0$ + **HMID**$_0$ would subsume **LTC**$_i$ ($i \geq 0$) since we can internally define each P_i and T by half-monotone inductive definitions.

This approach (indicated by P. Dybjer) seems quite interesting. In fact, we plan to move our results to M. Sato's Reflective Proof Theory, which is closely related to **LTC**. This work in detail is for future work.

4.6.4 Interpretation of Martin-Löf's type theory

We can also interpret Martin-Löf's type theory **ITT** in our theory.

Since **ITT** can be interpreted by **LTC**[3], we can indirectly interpret **ITT** in our theory by using **LTC** as an intermediate theory. However, we can give a direct interpretation of **ITT** in our theory. The technique is just the same as in the previous section for **LTC**; we first define how all the terms in **ITT** is translated. We then define a predicate which means the set of (translated) terms by induction. Next, we define predicates which mean "A is a type", and "a is a member of a type A" simultaneously. Here, we have negative occurrences of the predicate variable corresponding to 'a is a member of a type A" when we interpret Σ and Π types, but it is treated just the same as in the definition of A_1 and B_1 for **LTC**. Finally, we can interpret a first universe in the same way as A_2 and B_2 for **LTC**. We can repeat this construction at finitely many times to interpret **ITT**$_i$ for each $i \geq 0$.

We also have this interpretation is sound; namely if a judgement J is provable in **ITT**$_i$, we can prove its translation in **BT+HMID**$_0$.

Here, we omit the details for the lack of space.

4.7 Conclusion

We have studied a general form of half-monotone inductive definition in the context of a type-free first-order theory. It is an extension of monotone, simultaneous inductive definition, and is indispensable for *defining* provability relation within a theory, rather than a built-in relation. A characteristic point of this form is to allow a negative occurrence of a predicate variable which is being defined.

We changed the order between (tuples of) predicates so that such a kind of inductive definition is still monotone. The order is a slight modification of one proposed by Aczel[2], Hayashi and Nakano[21]. What is new in this chapter is to give a general theory of such styles of inductive definitions, and also give a sound realizability interpretation to show our theory is really constructive. We have shown that the Logical Theory of Constructions[3] can be interpreted in our theory quite naturally. We have also shown that provability relations which are useful for program improvement are definable in our theory. Our formulation of **HMID**$_0$ is sufficient for this purpose. As an application, we can prove equivalence of two definitions of provability relations, and can extract optimization programs by virtue of the program extraction theorem.

Although there are many related works, we briefly compare our work with two of them, which, we think, are most closely connected.

Comparison with Allen's Work

Allen[4, 5] interpreted Martin-Löf's type theory with universes in a type-free framework. He inductively defined a binary predicate τ which takes a term T and a binary predicate ϕ. Intuitively, $\tau(T, \phi)$ means "T is a type, and ϕ is an equivalence relation on the type T" (therefore a partial equivalence relation on the whole domain). For some fixed τ, the membership relation $a \in A$ in Martin-Löf's type theory will be interpreted by $\tau(\dot{A}, \phi) \wedge \phi(\dot{a}, \dot{a})$ for some ϕ where A and a are interpreted by $\dot{A}$ and $\dot{a}$. Having this interpretation in mind, τ is inductively defined. This inductive definition does not contain any negative occurrence of predicate variables being defined, hence is monotone in the usual sense. Allen constructed a model of Martin-Löf's type theory using this inductive definition. Since ϕ is a predicate over individuals (terms), τ is a second-order predicate, hence the whole theory becomes impredicative. Smith[40] formalized Allen's interpretation in **CTTR**, Martin-Löf's type theory with recursive types.

Allen's technique is superior to our theory in that he did not need any extra condition other than monotonicity while we need **HMONO-Q** condition. As a result, his theory might be more elegant if the only purpose is the interpretation of Martin-Löf's type theory. However, we believe that, our predicative theory has its own right because of the following reasons.

(1) In order to formalize something, we always look for as weak a theory as possible. Since a first-order theory with a least fixpoint operator (mu-calculus) is strictly weaker than the second order calculus[31], our first-order theory with half-monotone inductive definitions is strictly weaker than Allen and Smith's higher order framework. Our theory is therefore preferable in this respect.

(2) We believe that the form of our inductive definition is more natural and easier to understand than theirs.

In our theory, $\texttt{Proves}(a, A)$ is directly defined by a half-monotone inductive definition, while in Allen's interpretation, $a \in A$ is defined as "$\tau(A, \phi) \wedge \phi(a, a)$ for some ϕ" where τ is defined by a monotone inductive definition (for a higher order predicate).

Comparison with Dybjer's Work

Dybjer[15, 16] had a motivation which is similar to ours. He extended his recursive-induction mechanism[17] so that the universe hierarchy becomes definable in Martin-Löf-style type theory. As he mentioned, it can also incorporate the definition of ⟨Prop, Proves⟩. In some respects, his method is more flexible than ours, since he allows arbitrary recursive function for describing the second predicate (x in Proves(x, y)) while we stick to make all definitions inductive, and also he required no extra conditions like **HMONO-Q** (syntactic conditions are necessary). His method was possible because, in Martin-Löf-style type theories, a clause in each Introduction Rule always introduces a constructor symbol, so that the domain could never be overlapped through the process of inductive generation. We are working in a type-free theory, and the inductive definition mechanism is applicable to a wider range. As a result, we have to confine ourselves to inductive definitions, rather than inductive-recursive definitions. Besides this point, the differences between his work and ours are that (1) his theory is based on type theory while our base theory is untyped. (2) He allows only strictly-positive occurrences for the first predicate variable in his inductive definition, while we allow more general one (monotone inductive definition with **HMONO-Q** condition).

Dybjer[16] also gave a model for his theory. It is an interesting future problem to study the relationship of his model construction and our method.

Concluding Remarks

This work has been done in connection to Sato's $\mathcal{RPT}/\Lambda$ work[34]. $\mathcal{RPT}$ (Reflective Proof Theory) is a type-free constructive theory, but its spirit is very close to Martin-Löf's type theory and Frege structures. It has two judgements Prop(p) and Proves(a, p) [4]. If we introduce an **HMID**-like mechanism into $\mathcal{RPT}$, we would be able to define ⟨Prop, Proves⟩ in $\mathcal{RPT}$, and can extend the theory of program improvement in $\mathcal{RPT}$. We plan to amalgamate our induction mechanism into $\mathcal{RPT}$ smoothly.

[4]The original system had an infinite hierarchy of $Prop_i$ and $Proves_i$ where i ranges over ordinals, but we omit the subscripts here.

Chapter 5

Conservativeness of Λ! over λσ-calculus

Λ! is a unique functional programming language which has the facility of the *encapsulated assignment*, without losing *referential transparency* [34]. The `let`-construct in Λ! can be considered as an environment, which has a close relationship to the substitution in the calculus for explicit substitution λσ-calculus in [1].

This chapter discusses the relationship between these two calculi. We first define a slightly modified version of Λ! which adopts de Bruijn's index notation. We then define an injective map from λσ-calculus to Λ!, and show that the *Beta*-reduction and the σ-reductions in λσ-calculus correspond to the β-reduction and `let`-reductions in Λ!, respectively. Finally, we prove that, as equality theories, Λ! is *conservative* over λσ-calculus.

5.1 Introduction

Λ! is a unique functional programming language which has the facility of the *encapsulated assignment*, without losing *referential transparency* [34]. We can *assign* a value to a variable in a similar way as imperative programming languages. By this facility, Λ! programs can be quite efficient compared with programs written in ordinary functional programming languages. In spite of the existence of assignment, Λ! does not lose mathematically good features. Namely, it has a clear semantics, and it is referentially transparent in the sense that the equality is preserved through substitution. (See [34] for details.) We believe that Λ! is a good starting point of treating assignment in a mathematically well-founded manner.

In Λ!, the `let`-construct plays a fundamental role. The evaluation of the `let`-construct (`let` ((x a)) b) can be naturally considered as evaluating b under the environment $x = a$. This concept of environment is closely related to substitution in λσ-calculus[1]. λσ-calculus is an extension of λ-calculus where substitution has its own syntax, and explicitly described. λσ-calculus is mathematically well founded, since it is conservative over λ-calculus.

This chapter discusses the relationship between Λ! and $\lambda\sigma$-calculus. First, we define a slightly modified version of Λ!. The version we present in this chapter adopts de Bruijn's index notation, and has a slightly extended `let`-reductions compared with the original definition given by Sato[36]. Next, we define an injective map Φ from $\lambda\sigma$-calculus to Λ!. Then, we show that the *Beta*-reduction and the σ-reductions in $\lambda\sigma$-calculus correspond to the β-reduction and `let`-reductions in Λ!. Finally, we prove that, as equality theories, Λ! is *conservative* over $\lambda\sigma$-calculus. Namely, we have that $s = t$ in $\lambda\sigma$-calculus if and only if $\Phi(s) = \Phi(t)$ in Λ!.

In the following, we use metavariables t, s, u for $\lambda\sigma$-terms, θ, ϕ, χ for $\lambda\sigma$-substitutions, a, b, c for Λ!-terms, n, m for natural numbers.

5.2 $\lambda\sigma$-calculus with de Bruijn index

We quote the untyped $\lambda\sigma$-calculus in de Bruijn's index notation from [1]. We assume that readers are familiar with de Bruijn's notation and $\lambda\sigma$-calculus. See also [1] and [13].

Definition 30 (Term t and Substitution θ)

$$\begin{aligned} t &::= \mathtt{1} \mid tu \mid \lambda t \mid t\mathtt{[}\theta\mathtt{]} \\ \theta &::= \mathtt{id} \mid \uparrow \mid t\cdot\theta \mid \theta\circ\phi \end{aligned}$$

In de Bruijn's notation, all the bound variables disappear if they are just after λ, or otherwise replaced by indices `1`, `2`, $\cdots$. The indices represents the number of λ-binders between the occurrence of the bound variable and the λ-binder which actually binds this occurrence. For example, the term $\lambda x.\lambda y.xy$ will be represented by $\lambda(\lambda(21))$ in this notation. The term `1` represents the first index. An index larger than `1` is represented by combination of `1` and $\uparrow$. The terms tu and λt are as usual except that there appears no bound variable after λ. The term $t\mathtt{[}\theta\mathtt{]}$ is the term t to which the substitution θ is applied.

Each substitution intuitively represents a simultaneous substitution for indices. The substitution `id` is the identity substitution. The substitution $\uparrow$ is the "shift" operator, which substitutes $n+1$ for each index n. The substitution $t\cdot\theta$ is "cons" of a term t and a substitution θ, which intuitively represents the substitution $\{\mathtt{1} := t, \mathtt{2} := s_1, \mathtt{3} := s_2, \cdots\}$ where θ means $\{\mathtt{1} := s_1, \mathtt{2} := s_2, \cdots\}$ Finally, $\theta\circ\phi$ is the composition of two substitutions.

Definition 31 (Context C)

$$\begin{aligned} C &::= \langle\rangle \mid Ct \mid tC \mid \lambda C \mid C\mathtt{[}\theta\mathtt{]} \mid t\mathtt{[}\Theta\mathtt{]} \\ \Theta &::= \langle\rangle \mid C\cdot\theta \mid t\cdot\Theta \mid \Theta\circ\theta \mid \theta\circ\Theta \end{aligned}$$

A context C has just one hole $\langle\rangle$. To emphasize it, we sometimes use the notation $C\langle\rangle$. We may replace the hole with a term t or a substitution θ in a context $C\langle\rangle$, which is denoted as $C\langle t\rangle$ or $C\langle\theta\rangle$.

Definition 32 (1-step reduction $\to$)

Beta	$(\lambda t)s \to t\mathtt{[}s\cdot\mathtt{id}\mathtt{]}$
VarID	$\mathtt{1[id]} \to \mathtt{1}$
VarCons	$\mathtt{1[}t\cdot\theta\mathtt{]} \to t$
App	$(ts)\mathtt{[}\theta\mathtt{]} \to (t\mathtt{[}\theta\mathtt{]})(s\mathtt{[}\theta\mathtt{]})$
Abs	$(\lambda t)\mathtt{[}\theta\mathtt{]} \to \lambda(t\mathtt{[1}\cdot(\theta\circ\uparrow)\mathtt{]})$
Clos	$t\mathtt{[}\theta\mathtt{]}\mathtt{[}\phi\mathtt{]} \to t\mathtt{[}\theta\circ\phi\mathtt{]}$
IdL	$\mathtt{id}\circ\theta \to \theta$
ShiftId	$\uparrow\circ\,\mathtt{id} \to \uparrow$
ShiftCons	$\uparrow\circ(t\cdot\theta) \to \theta$
Map	$(t\cdot\theta)\circ\chi \to t\mathtt{[}\chi\mathtt{]}\cdot(\theta\circ\chi)$
Ass	$(\theta\circ\phi)\circ\chi \to \theta\circ(\phi\circ\chi)$

Rules other than *Beta* are called σ-rules or σ-reductions. Reduction relations for the *Beta*-rule and the σ-rules are written as $\to_B$ and $\to_\sigma$. *Beta*-reduction corresponds to the usual β-reduction in λ-calculus, but it does not actually perfom the substitution. It merely adds a new substitution $s\cdot\mathtt{id}$ to the term t. This substitution will be later resolved by σ-reductions.

Definition 33 (Reduction $\twoheadrightarrow$) *The relation $\twoheadrightarrow$ is the least relation satisfying the following conditions:*

1. *$\twoheadrightarrow$ is reflexive and transitive.*

2. *$t \to s$ implies $C\langle t\rangle \twoheadrightarrow C\langle s\rangle$.*

3. *$\theta \to \phi$ implies $D\mathtt{[}\theta\mathtt{]} \twoheadrightarrow D\mathtt{[}\phi\mathtt{]}$.*

The equality $=$ is the equivalence relation induced by $\twoheadrightarrow$.

Theorem 20 (Abadi et al) *The σ-reduction is confluent and terminating. The* $\lambda\sigma$-calculus *is confluent.*

The σ-normal form of a $\lambda\sigma$-term t is the normal form of t under the σ-reductions, and is written as $\sigma(t)$.

5.3 Λ! and plet-calculus

5.3.1 A Functional Programming Language Λ!

Λ! is a type-free functional programming language which has the facility of the *encapsulated assignment*. We can *assign* a value to a variable in a similar way with imperative languages. In spite of the existence of assignment, Λ! does not lose mathematically good features. Namely, it has a clear semantics by Church-Rosser Theorem, and it is referentially transparent in the sense that the equality is preserved through substitution. In this chapter, we reinforce this viewpoint by the fact that Λ! is a conservative extension of $\lambda\sigma$-calculus. The terms in $\lambda\sigma$-calculus can be naturally translated into Λ!, however, it is not clear that the equality is preserved through this translation, since the introduction of assignment to Λ! forces us to fix evaluation order to some extent while $\lambda\sigma$-calculus allows strong reductions, which may reduce subterms inside λ in an arbitrary context. Therefore, conservativeness of Λ! over $\lambda\sigma$-calculus is an interesting problem.

5.3.2 Modification to Λ!

This subsection explains the two different points between the original version of Λ! and the modified version used in this paper.

The first difference is that we use de Bruijn's index notation in the modified version, while variable names were used in the original version. In the new version, a variable is represented as a natural number (an index).

The other modification is explained below. Consider the following equation (taken from [34]):

```
(let ((x t)) (apply a b))
   = (apply (let ((x t)) a) (let ((x t)) b))
```

In this example, t, a, and b represent some terms in Λ!. In a natural translation from $\lambda\sigma$-calculus, this equation is expected to hold in any context. If t does not have assignable variables, the equation (without any context) holds. However, this equation does not hold in an arbitrary context. Consider the following equation (which is incorrect in the original Λ!):

```
(lambda (y) (let ((x y)) (apply a b)))
   = (lambda (y) (apply (let ((x y)) a) (let ((x y)) b)))
```

In the original Λ!, we have no way to evaluate the subterm `(let ((x y)) (apply a b))`, since y is not closed. From the Church-Rosser Property of the original Λ!, we can show that the equation above does not hold, which means the original version is not conservative over $\lambda\sigma$-calculus under a natural translation. It follows that the original Λ! is not conservative over the pure λ-calculus;

two equal λ-terms $\lambda y.(\lambda x.xx)y$ and $\lambda y.yy$ are translated into two Λ!-terms which are not equal[1].

This example motivates our modification. We allow reductions of a term `(let ((x t) a)`, not only in the case that t is closed (and a is arbitrary), but also in the case that t and a are read-only. A read-only term is a term which does not have side-effect. Note that a read-only term a may contain assignment even if a has assignment. In this case, every variable in the assignment must be bound by `let`-construct or `lambda`-construct in a. By extending `let`-reduction in this way, we can reduce, for example, the term like:

```
(lambda (y) (let ((x y)) (apply a b)))
   → (lambda (y) (apply a_x[y] b_x[y]))
```

where $a_x[y]$ means the usual substitution if a and b are read-only. We can show that the resulting calculus still satisfies the Church-Rosser Property, and has the referential transparency. We simply call this modified version Λ!, and use the term the "original" version if we mention Λ! in [34].

5.3.3 Definition of Λ!

The set of N-terms is defined for each natural number N as follows:

Definition 34 (Term a_N of Λ!)

```
a_N ::= n   if n ≥ 1
      | (set! n a_N)   if n ≤ N
      | (let ((a_N)) a_{N+1})
      | (while a_N a_N a_N)
      | (if a_N a_N a_N)
      | nil
      | (null? a_N)
      | (pair a_N a_N)
      | (pair? a_N)
      | (car a_N)
      | (cdr a_N)
      | (lambda () a_1)
      | (fun? a_N)
      | (apply a_N a_N)
      | (mu a_N)
```

[1]It follows that Theorem 4.6 in [34] also needs the modification to the definition of Λ!.

Intuitively, an N-term a_N is a term whose assignable variables are less than or equal to N. We sometimes call an N-term simply as a term. The term `(set!` n a_N`)` represents assignment for the variable n to the value a_N. In order to keep referential transparency, we restrict the assignable variables to be bound by a `let`-construct or a `lambda`-construct. Term constructs such as `while`, `if`, `nil`, `pair`, `car`, `cdr`, `lambda`, and `apply` have usual meaning. Terms such as `(null?` a`)` are predicates which decide whether a is `nil` or not, and return `true` or `false`. The term `(mu` a`)` is the μ-operator which invokes a recursive call. A term which is constructed from variables, `lambda`-construct, `apply`-construct, and `let`-construct is called a *pure* term. The terms `nil`, `(pair` a_N b_N`)`, and `(lambda ()` a_N`)` are called constructor terms, and the terms `(null?` a_N`)`, `(pair?` a_N`)`, and `(fun?` a_N`)` are called recognizer terms. We also say that `nil` and `(null?` a_N`)` are of the same kind. Likewise, `(pair` a_N b_N`)` and `(pair?` a_N`)`, are of the same kind, and `(lambda ()` a_N`)` and `(fun?` a_N`)` are of the same kind. Other combinations of pairs of these terms are of different kinds.

The reduction rules of Λ! are listed in the Appendix of this chapter. The confluency of the original Λ! was proved in [34], and the confluency of this modified version is proved similarly. The equality in Λ! is the least equivalence relation which contains $\rightarrow$. Instead of explaining the reductions rules in detail, we give a simple example here. Readers are encouraged to read [34] for thorough understanding of the original Λ!.

Example 1 (Reduction in Λ!) *Let t be the following term.*

```
(lambda ()
   (apply
      (apply
         (lambda () (lambda () (pair 1 (pair 2 3))))
         1)
      nil))
```

If we use the notation with variable names, t is written as follows:

```
(lambda (x)
   (apply
      (apply
         (lambda (y) (lambda (z) (pair z (pair y x))))
         x)
      nil))
```

The following sequence is a reduction sequence starting from t.

```
t  →  (by Rule 12)
(lambda ()
```

```
   (apply
      (let ((1))
         (lambda () (pair 1 (pair 2 3))))
      nil))
   →  (by Rule 17)
(lambda ()
   (apply
      (let ((1))
         (lambda () (pair 1 (pair 3 3))))
      nil))
   →  (by Rule 16)
(lambda ()
   (apply
      (lambda () (pair 1 (pair 2 2)))
      nil))
   →  (by Rule 12)
(lambda ()
   (let ((nil))
      (pair 1 (pair 2 2))))
   →  (by Rule 17)
(lambda ()
   (let ((nil))
      (pair nil (pair 2 2))))
   →  (by Rule 16)
(lambda () (pair nil (pair 1 1)))
```

In this chapter, we are mainly concerned with the fragment of Λ! consisting of pure terms, which are sufficient for the translation from $\lambda\sigma$-calculus. The fragment is called the *pure-fragment*. The pure-fragment is closed under reduction.

In the translation given later, we will need an intermediate calculus, which we temporarily call `plet`-calculus (parallel-`let` calculus).

Definition 35 (Term a of `plet`-calculus)

$$\begin{array}{lll} a & ::= & n \quad \textit{if } n \geq 1 \\ & | & \texttt{(let ((}a_1\texttt{) (}a_2\texttt{)} \cdots \texttt{(}a_k\texttt{))}\ b\texttt{)} \\ & | & \texttt{(lambda ()}\ a\texttt{)} \\ & | & \texttt{(apply}\ a\ b\texttt{)} \end{array}$$

Since `plet`-calculus is solely used for the translation, we do not define reduction rules for it.

5.4 Translation of $\lambda\sigma$-calculus into the pure-fragment

5.4.1 Translation of $\lambda\sigma$-calculus into plet-calculus

This subsection presents a translation from $\lambda\sigma$-terms to plet-terms. We begin with an auxiliary definition.

Definition 36 (Degree $\delta(a)$) *For each* plet-*term a, its degree $\delta(a)$ is a natural number defined as follows:*

$$\begin{aligned}
\delta(n) &\triangleq n \quad \textit{if } n \geq 1 \\
\delta(\texttt{(let ((}a_1\texttt{)} \cdots \texttt{(}a_k\texttt{))}\ b\texttt{))} &\triangleq max(\delta(a_1), \cdots, \delta(a_k), \delta(b) - k) \\
\delta(\texttt{(lambda () }a\texttt{))} &\triangleq max(1, \delta(a) - 1) \\
\delta(\texttt{(apply }a\ b\texttt{))} &\triangleq max(\delta(a), \delta(b))
\end{aligned}$$

Intuitively, $\delta(a)$ is the maximum index of free variables in a. If a does not have free variables, $\delta(a)$ is defined to be 1 rather than 0.

Definition 37 (Translation $t^\dagger$) *For each $\lambda\sigma$-term t, a* plet-*term a is defined as follows:*

$$\begin{aligned}
1^\dagger &\triangleq \texttt{1} \\
(ts)^\dagger &\triangleq \texttt{(apply }t^\dagger\ s^\dagger\texttt{)} \\
(\lambda t)^\dagger &\triangleq \texttt{(lambda () }t^\dagger\texttt{)} \\
(t[\theta])^\dagger &\triangleq \texttt{(let }\theta^{(\delta(t^\dagger))}\ t^\dagger\texttt{)}
\end{aligned}$$

The translation for substitution $-^{(n)}$ is defined as follows.

Definition 38 (Translation $\theta^{(n)}$) *For a substitution θ in $\lambda\sigma$-calculus and a natural number n $(n \geq 1)$, $\theta^{(n)}$ is a list of singleton-lists of* plet-*terms defined as follows:*

$$\begin{aligned}
\texttt{id}^{(n)} &\triangleq \texttt{((1) (2)} \cdots \texttt{(}n\texttt{))} \\
\uparrow^{(n)} &\triangleq \texttt{((2) (3)} \cdots \texttt{(}n+1\texttt{))} \\
(a \cdot \theta)^{(n)} &\triangleq \texttt{((}a^\dagger\texttt{) (}b_1\texttt{) (}b_2\texttt{)} \cdots \texttt{(}b_k\texttt{) (}k+2\texttt{))} \\
& \quad \textit{if } \theta^{(n)} \textit{ is } \texttt{((}b_1\texttt{) (}b_2\texttt{)} \cdots \texttt{(}b_k\texttt{))} \\
(\theta \circ \phi)^{(n)} &\triangleq \texttt{(((let }\phi^{(m)}\ a_1\texttt{))} \cdots \texttt{((let }\phi^{(m)}\ a_k\texttt{)))} \\
& \quad \textit{if } \theta^{(n)} \textit{ is } \texttt{((}a_1\texttt{) (}a_2\texttt{)} \cdots \texttt{(}a_k\texttt{))} \\
& \quad \textit{and } m \textit{ is } max(\delta(a_1), \cdots, \delta(a_k)).
\end{aligned}$$

Proposition 3 *The translation $-^\dagger$ is injective.*

Proof. First note that, for each substitution θ and natural number n, the length (as a list) of $\theta^{(n)}$ is equal to or more than n. It follows that the length of $(a \cdot \theta)^{(n)}$ is more than $n+1$, so $(a \cdot \theta)^{(n)}$ cannot be identical to $\texttt{id}^{(n)}$ nor $\uparrow^{(n)}$. Moreover, its last element is a natural number $k+2$, and differs[2] from the last element of $(\theta \circ \phi)^{(n)}$. Hence, the images of $-^{(n)}$ for four classes of substitutions do not overlap. Using this fact, we can prove that, $t^\dagger$ and $\theta^{(n)}$ (for each n) are injective by the simultaneous induction on the complexity of the term t and the substitution θ. □

5.4.2 Translation from plet-calculus to Λ!

First we define a^+ for each plet-term a. Intuitively, a^+ is the term a with each free variable shifted (added by one), for example,

$$\texttt{(apply 3 (lambda () (apply 1 2)))}^+$$

is (apply 4 (lambda () (apply 1 3))). To define a^+, we need to define an auxiliary function a^+_m, which adds one for each free variable in a whose value is more than m.

Definition 39 (a^+_m)

$$\begin{aligned}
n^+_m &\triangleq n \quad \textit{if } n \leq m \\
n^+_m &\triangleq n+1 \quad \textit{if } n > m \\
\texttt{(let ((}a_1\texttt{)} \cdots \texttt{(}a_k\texttt{))}\ b\texttt{)}^+_m &\triangleq \texttt{(let (((}a_1\texttt{)}^+_m\texttt{)} \cdots \texttt{((}a_k\texttt{)}^+_m\texttt{))}\ b\texttt{)} \\
\texttt{(lambda () }a\texttt{)}^+_m &\triangleq \texttt{(lambda () }a^+_{m+1}\texttt{)} \\
\texttt{(apply }a\ b\texttt{)}^+_m &\triangleq \texttt{(apply }a^+_m\ b^+_m\texttt{)}
\end{aligned}$$

We simply write a^+_0 as a^+.

Definition 40 (Translation * from plet-calculus to Λ!)

$$\begin{aligned}
n^* &\triangleq n \\
\texttt{(let () }b\texttt{)}^* &\triangleq \texttt{(let (}b^*\texttt{) 1)} \\
\texttt{(let ((}a_1\texttt{)} \cdots \texttt{(}a_{k-1}\texttt{) (}a_k\texttt{))}\ b\texttt{)}^* &\triangleq \texttt{(let ((}a_k\texttt{))}\ c\texttt{)} \\
& \quad \textit{if } c \textit{ is } \texttt{(let ((}a_1{}^+\texttt{)} \cdots \texttt{(}a_{k-1}{}^+\texttt{))}\ b\texttt{)}^* \\
\texttt{(lambda () }a\texttt{)}^* &\triangleq \texttt{(lambda () }a^*\texttt{)} \\
\texttt{(apply }a\ b\texttt{)}^* &\triangleq \texttt{(apply }a^*\ b^*\texttt{)}
\end{aligned}$$

[2]This is the reason why we attached the (meaningless) term $k+2$ in the definition.

In the following, we sometimes regard `plet`-terms as pure terms in Λ! (through the translation $*$).

The translation Φ from $\lambda\sigma$-calculus to the pure-fragment is defined as follows.

Definition 41 (Translation Φ from $\lambda\sigma$-calculus to Λ!)

$$\Phi(t) \triangleq (t^\dagger)^*$$

Theorem 21 *$-^*$ is injective. Hence Φ is injective.*

Proof. Clear.

Remark 3 *If a non-injective map were allowed as the translation Φ, part of our results (the first part of Theorem 22) would become trivial as shown below.*

In $\lambda\sigma$-calculus, the set of σ-normal forms can be regarded as the set of pure λ-terms, so the map $\sigma(\_)$ can be regarded as the translation from $\lambda\sigma$-terms to pure λ-terms. We have that, $s = t$ holds if and only if $\sigma(s) = \sigma(t)$ holds. Λ! is also conservative over pure λ-calculus [3]. *Namely, there is a map Ψ from pure λ-terms to Λ!-terms such that $a = b$ holds if and only if $\Psi(a) = \Psi(b)$ holds. Let Φ be the composition of σ and Ψ; then we have that $s = t$ holds in $\lambda\sigma$-calculus if and only if $\Phi(s) = \Phi(t)$ holds in Λ!.* □

5.4.3 Properties of the translation Φ

Here, we prove that the translation Φ preserves the equality. First we state an extension of Lemma 4.2 in [34].

Lemma 8 *Let a and b be pure N-terms for some natural number N.*

Then, the term `(let ((`a`))` b`)` *reduces to $b\{1 := a, 2 := 1, 3 := 2, \cdots, k := k-1\}$ using* `let`*-rules only, where k is $\delta(b)$.*

Here $\{1 := a, 2 := 1, 3 := 2, \cdots, k := k-1\}$ denotes the simultaneous substitution. Note that a and b are not necessarily `0`-terms. As was stated in Section 5.3.2, this lemma does not hold for the original Λ!, since we cannot reduce the term `(let ((`a`))` b`)` if a is not closed. On the contrary, the version we present in this chapter satisfies this lemma, since all pure N-terms are read-only, which enables us to reduce the term `(let ((`a`))` b`)`.

Similarly, we have the following lemma.

Lemma 9 *Let $a_1 \cdots a_k$ and b be pure N-terms, and k be $\delta(b)$. If $n \geq k$, then* `((let ((`a_1`)` $\cdots$ `(`a_n`))` b`)` *reduces to $b\{1 := a_1, 2 := a_2, \cdots, k := a_k\}$ using* `let`*-rules only.*

[3]This claim does not hold for the original Λ!, but it does hold for the modified version presented in this chapter.

Note that, we regard `plet`-terms as Λ!-terms through the translation $(\,)^*$ in Lemma 9.

These lemmas are proved by the induction on b.

Proposition 4 *For each term-reduction rule $t \rightarrow s$ in $\lambda\sigma$-calculus, $\Phi(t) = \Phi(s)$ holds in Λ!. For each substitution-reduction rule $\theta \rightarrow \phi$ and a term s in $\lambda\sigma$-calculus, $\Phi(s[\theta]) = \Phi(s[\phi])$ holds in Λ!.*

Proof. This proposition is proved by the case-analysis.

(Beta) The left hand side (LHS, in short) of *Beta* is translated into

```
(apply (lambda () $t^\dagger$) $s^\dagger$)
```

which β-reduces to

```
(let (($s^\dagger$)) $t^\dagger$)
```

By Lemma 8, this is equal to $t^\dagger\{1 := s^\dagger, 2 := 1, 3 := 2, \cdots\}$.

The right hand side (RHS, in short) of *Beta* is translated into

```
(let (($s^\dagger$) (1) ··· ($n-1$)) $t^\dagger$)
```

Calculation of indexes shows that this is equal to $t^\dagger\{1 := s^\dagger, 2 := 1, 3 := 2, \cdots\}$.

(VarID) LHS is translated into `(let ((1)) 1)` which reduces to `1`. RHS is translated into `1`.

(VarCons) LHS is translated into `(let ((`$t^\dagger$`)` $\cdots$`) 1)` which reduces to $t^\dagger$. RHS is translated into $t^\dagger$.

(App) Suppose $\theta^{(\delta((ts)^\dagger))}$ is `((`a_1`)` $\cdots$ `(`a_k`))`.

LHS is translated into `(let ((`a_1`) (`a_2`)` $\cdots$ `(`a_k`)) (apply` $t^\dagger$ $s^\dagger$`))`.

RHS is translated into

```
(apply
  (let (($a_1$) ($a_2$) ··· ($a_l$)) $t^\dagger$)
  (let (($a_1$) ($a_2$) ··· ($a_m$)) $s^\dagger$))
```

where l and m are $\delta(t^\dagger)$ and $\delta(s^\dagger)$.

By Lemma 9, `(let ((`a_1`)` $\cdots$ `(`a_k`))` $t^\dagger$`)` is equal to

$t^\dagger$`{1:=`$a_1, \cdots, l$`:=`a_l`}`

and similarly for $s^\dagger$. Hence, by Lemma 8, LHS and RHS reduce to

```
(apply t†{1:=a₁,···,l:=aₗ}
       s†{1:=a₁,···,m:=aₘ})
```

(Abs) Suppose $\theta^{(\delta((\lambda t)^\dagger))}$ is `((`a_1`)` $\cdots$ `(`a_k`))`.

LHS is translated into `(let ((`a_1`)` $\cdots$ `(`a_k`)) (lambda ()` $t^\dagger$`))`. This reduces to

```
(lambda ()
   t†{2:=a₁{1:=2,2:=3,···},
      ...
      k + 1:=aₖ{1:=2,2:=3,···}}).
```

RHS is translated into

```
(lambda ()
  (let ((1)
        ((let ((2) (3) ···) a₁))
        ...
        ((let ((2) (3) ···) aₖ)))
    t†))
```

The latter term reduces to the former by Lemma 9.

(Clos) Suppose $\theta^{(\delta(t^\dagger))}$ is `((`a_1`)` $\cdots$ `(`a_l`))`, and $\phi^{(n)}$ is `((`b_1`)` $\cdots$ `(`b_k`))` where n is $max(a_1, \cdots, a_l)$.

LHS is translated into

```
(let ((b₁) ··· (bₖ))
  (let ((a₁) ··· (aₗ)) t†))
```

RHS is translated into

```
(let (((let ((b₁) ··· (bₖ)) a₁))
      ...
      ((let ((b₁) ··· (bₖ)) aₗ)))
  t†)
```

LHS and RHS reduce to

```
t†{1:=a₁{1:=b₁,···,k:=bₖ},
   ...
   l:=aₗ{1:=b₁,···,k:=bₖ}}.
```

(IdL) Let n be $\delta(s)$. Suppose $\theta^{(n)}$ is `((`b_1`)` $\cdots$ `(`b_k`))`. Then, s`[id`$\circ\theta$`]` is translated into

```
(let (((let ((b₁) ··· (bₖ)) 1))
      ...
      ((let ((b₁) ··· (bₖ)) n)))
  s†)
```

Since $k \geq n$, this reduces to `(let ((`b_1`)` $\cdots$ `(`b_n`))` $s^\dagger$`)` Then this term is identical to `(`s`[`θ`])`† by Lemma 9.

(ShiftId) Let n be $\delta(s)$. The term s`[`$\uparrow\circ$ `id]` is translated into

```
(let (((let ((1) ··· (n + 1)) 2))
      ((let ((1) ··· (n + 1)) 3))
      ...
      ((let ((1) ··· (n + 1)) n + 1)))
  s†)
```

This reduces to `(let ((2)` $\cdots$ `(`$n+1$`))` $s^\dagger$`)` which is identical to `(`s`[`$\uparrow$`])`†.

(ShiftCons) Let n be $\delta(s)$. Suppose $\theta^{(n+1)}$ is `((`b_1`)` $\cdots$ `(`b_k`))`.
Then, s`[`$\uparrow\circ(t\cdot\theta)$`]` is translated into

```
(let (((let ((t†) (b₁) ··· (bₖ) (k + 2)) 2))
      ((let ((t†) (b₁) ··· (bₖ) (k + 2)) 3))
      ...
      ((let ((t†) (b₁) ··· (bₖ) (k + 2)) n + 1)))
  s†)
```

We also have $k \geq n+1$, and the term above reduces to

```
(let ((b₁) ··· (bₙ)) s†)
```

which is equal to `(`s`[`θ`])`†.

(Map) Let n be $\delta(s)$. Suppose $\theta^{(n)}$ is `((`a_1`)` $\cdots$ `(`a_k`))`, and $\chi^{(m)}$ is `((`b_1`)` $\cdots$ `(`b_l`))` where m is $max(\delta(t), \delta(a_1), \cdots, \delta(a_k), k+2)$.

Then, s`[`$(t\cdot\theta)\circ\chi$`]` is translated into

```
(let (((let ((b_1) ··· (b_l)) t^†))
      ((let ((b_1) ··· (b_l)) a_1))
      ...
      ((let ((b_1) ··· (b_l)) a_k))
      ((let ((b_1) ··· (b_l)) k + 2)))
  s^†)
```

$s[t[\chi] \cdot (\theta \circ \chi)]$ is translated into

```
(let (((let ((b_1) ··· (b_o)) t^†))
      ((let ((b_1) ··· (b_q)) a_1))
      ...
      ((let ((b_1) ··· (b_q)) a_k))
      (k + 2))
  s^†)
```

where $\delta(t)$ is p, $\chi^{(p)}$ is $((b_1)\ \cdots\ (b_o))$, $max(\delta(a_1), \cdots, \delta(a_k))$ is r, and $\chi^{(r)}$ is $((b_1)\ \cdots\ (b_q))$.

We have that $p \leq o \leq l$, $r \leq q \leq l$, and $n \leq k$, therefore, by Lemma 9, both of these terms are equal to

```
s^†{1:=t^†{1:=b_1, ···, p:=b_p},
    2:=a_1{1:=b_1, ···, r:=b_r},
    ...
    n:=a_n{1:=b_1, ···, r:=b_r}}
```

(Ass) Let n be $\delta(s)$. Suppose $\theta^{(n)}$ is $((a_1)\ \cdots\ (a_k))$, $\phi^{(o)}$ is $((b_1)\ \cdots\ (b_l))$, and $\chi^{(p)}$ is $((c_1)\ \cdots\ (c_m))$ for appropriate o and p.

Then, $s[(\theta \circ \phi) \circ \chi]$ is translated into

```
(let (((let ((c_1) ··· (c_m))
          (let ((b_1) ··· (b_l)) a_1))
      ...
      ((let ((c_1) ··· (c_m))
          (let ((b_1) ··· (b_l)) a_k)))
  s^†)
```

$s[\theta \circ (\phi \circ \chi)]$ is translated into

```
(let (((let (((let ((c_1) ··· (c_m)) b_1))
              ...
              ((let ((c_1) ··· (c_m)) b_l)))
          a_1))
      ...
      ((let (((let ((c_1) ··· (c_m)) b_1))
              ...
              ((let ((c_1) ··· (c_m)) b_l)))
          a_k)))
  s^†)
```

Both of these terms reduce to

```
s^†{1:=a_1{1:=b_1{1:=c_1, ···, m:=c_m}, ···, l:=b_l{1:=c_1, ···, m:=c_m}},
    ...
    1:=a_k{1:=b_1{1:=c_1, ···, m:=c_m}, ···, l:=b_l{1:=c_1, ···, m:=c_m}}
```

Note that, we have used only `let`-rules for proving the cases for σ-rules. Note also that, the 1-step *Beta* reduction can be simulated by the 1-step β-reduction with some `let`-rules. □

Proposition 5 *Let t and s be $\lambda\sigma$-terms. If $t = s$, then $\Phi(t) = \Phi(s)$ in* Λ!.

Proof. We first prove that, the result of Proposition 4 can be extended to an arbitrary context. Namely, for a context $C\langle\rangle$, if $t \to s$, then $\Phi(C\langle t\rangle) = \Phi(C\langle s\rangle)$ where t and s are terms or substitutions. These are straightforward if the used reduction rule is a σ-rule. However, in the case of *Beta*-rule, there occurs a subtle point; for example, $((\lambda 2)3)[\uparrow]$ is translated into

```
(let ((2) (3) (4)) (apply (lambda () 2) 3)).
```

On the other hand, the result of applying *Beta* rule to it is $1[\uparrow]$ which is translated into `(let ((2)) 1)`. We can use Lemma 9 to overcome this difficulty, and can prove the equality of $\Phi(C\langle t\rangle)$ and $\Phi(C\langle s\rangle)$.

Finally, we can extend the result for 1-step reductions to the general case, and get the desired proposition. □

By checking the proofs, we know that, if $t = s$ is shown by σ-rules only, then $\Phi(t) = \Phi(s)$ is shown by `let`-rules only.

5.5 Translation of the pure-fragment into λσ-calculus

We now define the reverse translation, namely the translation from the pure-fragment of Λ! to $\lambda\sigma$-calculus.

Definition 42 (Translation Ψ)

$$\begin{aligned}
\Psi(n) &\triangleq \texttt{1[}\uparrow^n\texttt{]} \\
\Psi(\texttt{(let ((}a\texttt{))}\ b\texttt{)}) &\triangleq \Psi(b)\texttt{[}\Psi(a)\cdot\texttt{id]} \\
\Psi(\texttt{(lambda () }a\texttt{)}) &\triangleq \lambda\Psi(a) \\
\Psi(\texttt{(apply }a\ b\texttt{)}) &\triangleq \Psi(a)\Psi(b)
\end{aligned}$$

In the first clause, $\texttt{1[}\uparrow^n\texttt{]}$ is n-times application of substitution, that is, $\texttt{1}\overbrace{\texttt{[}\uparrow\texttt{]}\cdots\texttt{[}\uparrow\texttt{]}}^{n}$.

Proposition 6 *We have the following;*
1. Ψ is injective.
2. Let a and b be pure N-terms in Λ!. If $a = b$, then $\Psi(a) = \Psi(b)$ holds.

Proof. We first prove the theorem for the case of $a \to b$. It is proved by the induction on the derivation of $a \to b$. We only have to consider Rules 1, 3, 5, 7, 12, 16, 17, and 19.

(Rules 1, 3, 5, 7) These cases are proved easily.

(Rule 12) Suppose a is `(apply (lambda () c) d)`, b be `(let ((d')) c')`, $c \to c'$ and $d \to d'$. Then, $\Psi(a)$ is $(\lambda\Psi(c))\Psi(d)$, and $\Psi(b)$ is $\Psi(c')\texttt{[}\Psi(d')\cdot\texttt{id]}$. By the induction hypothesis and the *Beta* rule in λσ-calculus, these terms are equal.

(Rule 16) Suppose a is `(let ((c)) d)`, $\texttt{1} \notin FV(d)$. $d \to e$, and b is e^-. We have $\Psi(a)$ is $\Psi(d)\texttt{[}\Psi(c)\cdot\texttt{id]}$. We can show that, all the occurrences of `1` in $\Psi(d)$ are followed by one or more ↑'s, hence $\Psi(d)\texttt{[}\Psi(c)\cdot\texttt{id]}$ is equal to $\Psi(d)\{2 := \texttt{1}, 3 := \texttt{2}, \cdots\}$, which is again equal to $\Psi(d^-)$. We have $\texttt{1} \notin FV(e)$, and, therefore, $d^- \to e^-$. By the induction hypothesis $\Psi(d^-) = \Psi(e^-)$, hence $\Psi(a) = \Psi(b)$.

(Rule 17) Suppose a is `(let ((c)) d)`, $\rho(d,p) = \texttt{1}$, $c \to c'$, $d \to d'$, and $e \equiv d'_p[c'^+]$, b is `(let ((c')) e)`. Then, $\Psi(a)$ is $\Psi(d)\texttt{[}\Psi(c)\cdot\texttt{id]}$, which is equal to $\Psi(d')\texttt{[}\Psi(c')\cdot\texttt{id]}$, by the induction hypothesis. By the induction on the term $\Psi(d')$, we have that the σ-normal forms of this term and the term $\Psi(e)\texttt{[}\Psi(c')\cdot\texttt{id]}$ are identical. Hence we have that $\Psi(a)$ and $\Psi(b)$ are σ-equal.

(Rule 19) This case is proved in a similar way as Rule 17.

We can extend the result above to the equality $a = b$. □

5.6 Main Theorem

This section presents the main theorem of this chapter.

Proposition 7 *For each λσ-term t, $\Psi(\Phi(t)) = t$ holds. Moreover, the equality is shown by the* `let`*-rules only.*

Proof. This proposition is proved by the induction on the term t. □

Theorem 22 *Let t and s be λσ-terms. Then, $t = s$ holds if and only if $\Phi(t) = \Phi(s)$ holds.*

Moreover, if t is shown to be equal to s using σ-rules only, $\Phi(t)$ and $\Phi(s)$ are shown to be equal using `let`*-rules only. If t is shown to be equal to s by several times applications of the Beta-rule, $\Phi(t)$ and $\Phi(s)$ are shown to be equal by the same times applications of the β-rule, and some applications of* `let`*-rules.*

Proof. The first part follows from Propositions 5, 6 and 7. The second part follows from the remarks for these propositions. □

Remark 4 *Theorem 22 shows that the pure-fragment of Λ! and λσ-*calculus *have a close relationship; as equality theories, Λ! (the version presented in this chapter) is conservative over λσ-*calculus.

*However, we can see several differences between them. Firstly, the reduction rules do not directly correspond, namely, $t \twoheadrightarrow s$ in λσ-*calculus *does not necessarily imply $\Phi(t) \to \Phi(s)$ in Λ!. Secondly, substitutions are objects in λσ-*calculus, *and can be directly treated, while its corresponding expression* `(let ((a)) ⟨⟩)` *is not a term in Λ!. This reflects that, in Λ! we only consider the environment with some term, and never treat one as an independent object. One of the design goals of Λ! is to treat assignment in a mathematically well founded manner, which means we want to keep the referential transparency in our sense, and therefore, we do not separate terms and environments.*

*As in λσ-*calculus, *we have a complete normal-order strategy for the reductions in Λ!, which we plan to implement on a computer.* □

5.7 Conclusion

We have shown the rigid relationship between "explicit substitution" (λσ-calculus) and our functional language Λ!. We first presented a modified version of Λ! so that we may reduce `let`-terms under the "read-only" condition. We used de Bruijn's index notation in this presentation. We then gave a translation from λσ-calculus into the pure-fragment of Λ!, and a reverse one. We proved that, through these translations, σ-rules correspond to `let`-rules, *Beta*-rule corresponds to β-rule, and finally Λ! is conservative over λσ-calculus. We also presented a brief sketch of translation for calculi with variable names. Together with the Church-Rosser property and the referential transparency presented in [34], our result establishes that Λ! is a well-founded programming language with assignment.

As a future work, we should extend RPT so that reasoning about Λ!-programs can be formalized in RPT, and then extend our Constructive Programming System to include such reasoning. By doing these things, we can synthesize Λ!-programs (with the assignment and the `while` statements) by the way of Constructive Programming.

Appendix: The Definition of Λ! in de Bruijn's notation

The Appendix gives several definitions including the reduction rules of Λ! in de Bruijn's index notation.

A *position* is a finite sequence of positive integers, with ϵ being the empty sequence. For instance, `121` is a position.

Each subterm in a term is specified by a position in a usual way. We use the notation t/p for the subterm of a term t at the position p. For instance,

> `(apply (apply` a b`)` c`)/22` is a, and
> `(apply (apply` a b`)` c`)/`ϵ is `(apply (apply` a b`)` c`)`.

For a term a and a position p, $\nu(a,p)$ intuitively means the number of surrounding binders (`let` or `lambda`) at the position p, and is defined as follows.

Definition 43

$$\begin{aligned}
\nu(\texttt{(let ((}b\texttt{))}\ c\texttt{)}, 211q) &\triangleq \nu(b,q)\\
\nu(\texttt{(let ((}b\texttt{))}\ c\texttt{)}, 3q) &\triangleq \nu(c,q)+1\\
\nu(\texttt{(lambda ()}\ b\texttt{)}, 3q) &\triangleq \nu(b,q)+1\\
\nu(\texttt{(}f\ b_1\ \cdots\ b_m\texttt{)}, iq) &\triangleq \nu(b_{i-1},q)\ \textit{where } f \textit{ is not } \texttt{let} \textit{ nor } \texttt{lambda},\\
\nu(a,p) &\triangleq 0\ \textit{otherwise}
\end{aligned}$$

Suppose a/p is a variable i. This occurrence of a variable is called *bound* if $i \leq \nu(a,p)$, and *free* otherwise.

Next, we define a natural number $\rho(a,p)$ for a term a and a position p. In a term a, there may be several occurrences of a variable, and each may take a different value. We, therefore, sometimes need to know the absolute value of a variable-occurrence if we look at this occurrence from outside of a. The number $\rho(a,p)$ is defined to be $i - \nu(a,p)$ where a/p is a free occurrence of a variable in a, and i is the variable. Otherwise, $\rho(a,p)$ is undefined.

$FV(a)$ is the set of $\rho(a,p)$ where p ranges over all the free occurrences of variables in a.

Let us take an example. Let a be `(let ((2)) (pair 1 2))`. Then $\nu(a,211)$ is 0, the occurrence at 211 of a is free, and $\rho(a,211)$ is 2. $\nu(a,32)$ is 1, the occurrence at 32 of a is bound, and $\rho(a,32)$ is undefined. $\nu(a,33)$ is 1, the occurrence at 33 of a is free, and $\rho(a,33)$ is 1.

For a term a, two terms a^+ and a^- are the term a with each free variable added by one, and subtracted by one, respectively. For instance,

> a^+ is `(let ((3)) (pair 1 3))`
> a^- is `(let ((1)) (pair 1 1))`

A precise definition of a^+ is given by Definition 39.

An N-term a is called *N-closed* if $FV(a) \cap \{1,\cdots,N\} = \emptyset$. The set C_N represents the set of N-closed terms. An N-term a is called *N-read-only* if, for any subterm in the form `(set!` n b`)`, n is bound in a. R_N represents the set of N-read-only terms.

Definition 44 (Substitution) *Let a and d be terms, and p be a position. We will define a term $a_p[d]$ as follows:*

- *If p is ϵ, $a_p[d]$ is d.*
- *Otherwise,*
  - *if a is* `(lambda ()` b`)`*,*
    then $a_p[d]$ is `(lambda ()` $b_q[d^+]$`)` *if p is* `3`*q, and is undefined otherwise.*
  - *if a is* `(let ((`b`))` c`)`*,*
    then $a_p[d]$ is `(let ((`$b_q[d]$`))` c`)` *if p is* `211`*q. $a_p[d]$ is* `(let ((`b`))` $c_q[d^+]$`)` *if p is* `3`*q. $a_p[d]$ is undefined otherwise.*
  - *if a is* `(`f b_1 $\cdots$ b_n`)` *where f is not* `let` *nor* `lambda`*,*
    $a_p[d]$ is `(`f b_1 $\cdots$ $b_{i_q}[d]$ $\cdots$ b_m`)` *if p is jq, $2 \leq j \leq m+1$, and i is $j-1$.*
  - *otherwise*
    $a_p[d]$ is undefined.

Substitution for multiple occurrences $a_{p_1,\cdots,p_k}[b]$ is defined to be $a_{p_1}[b]$ if $k=1$, and $(a_{p_1}[b])_{p_2,\cdots,p_k}[b]$ if $k>1$.

We next define the set $\Sigma_N(a)$ for each N-term a in Λ!. Intuitively, if $p \in \Sigma_N(a)$, the subterm a/p should be evaluated at the next step by the `let`-reduction. Note, however, that we fix the evaluation order only for one occurrence of the `let`-construct. If other rules are applicable, or there are other `let`-constructs which do not interfere with this `let`-construct, we may evaluate other subterms than one specified by $\Sigma_N(a)$. For a position p and a set S, pS is the set $\{pq \mid q \in S\}$.

Definition 45 *If $a \in C_N$, then $\Sigma_N(a)$ is $\emptyset$. Otherwise,*

$$
\begin{array}{rcl}
\Sigma_N(n) & \triangleq & \{\epsilon\} \\
\Sigma_N(\texttt{(let ((}a\texttt{)) }b\texttt{)}) & \triangleq & \begin{cases} 3\Sigma_{N+1}(b) & \textit{if } a \in C_N, \\ 211\Sigma_N(a) \cup 3\Sigma_{N+1}(b) & \textit{if } a \in R_N \textit{ and } b \in R_{N+1}, \\ 211\Sigma_N(a) & \textit{otherwise.} \end{cases} \\
\Sigma_N(\texttt{(set! }n\ a\texttt{)}) & \triangleq & \begin{cases} \{\epsilon\} & \textit{if } a \in C_N, \\ 3\Sigma_N(a) & \textit{otherwise.} \end{cases} \\
\Sigma_N(\texttt{(lambda () }a\texttt{)}) & \triangleq & \{\epsilon\} \\
\Sigma_N(\texttt{(while }a\ b\ c\texttt{)}) & \triangleq & 2\Sigma_N(a) \\
\Sigma_N(\texttt{(if }a\ b\ c\texttt{)}) & \triangleq & 2\Sigma_N(a) \\
\Sigma_N(\texttt{(apply }a\ b\texttt{)}) & \triangleq & \begin{cases} 3\Sigma_N(b) & \textit{if } a \in C_N, \\ 2\Sigma_N(a) \cup 3\Sigma_N(b) & \textit{if } a \in R_N \textit{ and } b \in R_N, \\ 2\Sigma_N(a) & \textit{otherwise.} \end{cases} \\
\Sigma_N(\texttt{(pair }a\ b\texttt{)}) & \triangleq & \begin{cases} 3\Sigma_N(b) & \textit{if } a \in C_N, \\ 2\Sigma_N(a) \cup 3\Sigma_N(b) & \textit{if } a \in R_N \textit{ and } b \in R_N, \\ 2\Sigma_N(a) & \textit{otherwise.} \end{cases} \\
\Sigma_N(\texttt{(f }a\texttt{)}) & \triangleq & 2\Sigma_N(b) \quad \textit{where } f \textit{ is a term construct not listed above}
\end{array}
$$

Note that, for a pure, open term a, $\Sigma_N(a)$ is not empty.

The 1-step reduction relation $\rightarrow$ in de Bruijn notation is defined as follows:

Definition 46 **1.** *If n is a variable (an index), then $n \rightarrow_N n$.*

2. *If $a \rightarrow_N d$ and s is one of* `fun?`, `null?`, `pair?`, `car`, `cdr`, *and* `mu`, *then*
$$\texttt{(}s\ a\texttt{)} \rightarrow_N \texttt{(}s\ d\texttt{)}$$

3. *If $a \rightarrow_N d$ and $b \rightarrow_N e$ and s is one of* `pair`, `apply`, *then*
$$\texttt{(}s\ a\ b\texttt{)} \rightarrow_N \texttt{(}s\ d\ e\texttt{)}$$

4. *If $a \rightarrow_N d$, $b \rightarrow_N e$ and $c \rightarrow_N f$ and s is* `if` *or* `while`, *then*
$$\texttt{(}s\ a\ b\ c\texttt{)} \rightarrow_N \texttt{(}s\ d\ e\ f\texttt{)}$$

5. *If $a \rightarrow_1 d$, then* $\texttt{(lambda () }a\texttt{)} \rightarrow_N \texttt{(lambda () }d\texttt{)}$

6. *If $a \rightarrow_N d$, then* $\texttt{(set! }n\ a\texttt{)} \rightarrow_N \texttt{(set! }n\ d\texttt{)}$

7. *If $a \rightarrow_N d$ and $b \rightarrow_{N+1} e$, then* $\texttt{(let ((}a\texttt{)) }b\texttt{)} \rightarrow_N \texttt{(let ((}d\texttt{)) }e\texttt{)}$

8. *If $a \in R_N$,* $\texttt{(s }a\texttt{)}$ *is a recognizer term of some kind, and a is a constructor term of the same kind, then* $\texttt{(s }a\texttt{)} \rightarrow_N \texttt{true}$.

9. *If $a \in R_N$,* $\texttt{(s }a\texttt{)}$ *is a recognizer term of some kind, and a is a constructor term of a different kind, then* $\texttt{(s }a\texttt{)} \rightarrow_N \texttt{false}$.

10. *If $a \in R_N$, $b \in R_N$, and $a \rightarrow_N d$, then* $\texttt{(car (pair }a\ b\texttt{))} \rightarrow_N d$.

11. *If $a \in R_N$, $b \in R_N$, and $b \rightarrow_N e$, then* $\texttt{(cdr (pair }a\ b\texttt{))} \rightarrow_N e$.

12. *If* $\texttt{(lambda () }a\texttt{)} \in R_N$, *$b \in R_N$, $a \rightarrow_{N+1} d$ and $b \rightarrow_N e$, then*
$$\texttt{(apply (lambda () }a\texttt{) }b\texttt{)} \rightarrow_N \texttt{(let ((}e\texttt{)) }d\texttt{)}$$

13. *If $a \in R_N$ and $a \rightarrow_N d$, then* $\texttt{(mu }a\texttt{)} \rightarrow_N \texttt{(apply }d\texttt{ (mu }d\texttt{))}$.

14. *If $b \rightarrow_N e$ then* $\texttt{(if true }b\ c\texttt{)} \rightarrow_N e$.

15. *If $c \rightarrow_N f$ then* $\texttt{(if false }b\ c\texttt{)} \rightarrow_N f$.

16. *If $a \in R_N$, $1 \notin FV(b)$, $b \in C_{N+1}$, and $b \rightarrow_{N+1} e$, then*
$$\texttt{(let ((}a\texttt{)) }b\texttt{)} \rightarrow_N e^-$$

17. *If $a \rightarrow_N d$, $p \in \Sigma_{N+1}(b)$, $\rho(b,p) = 1$, $b \rightarrow_{N+1} e$, and either (i) $a \in C_N$, or (ii) $a \in R_N$ and $b \in R_{N+1}$, then* $\texttt{(let ((}a\texttt{)) }b\texttt{)} \rightarrow_N \texttt{(let ((}d\texttt{)) }e_p[d^+]\texttt{)}$

18. *If $p \in \Sigma_{N+1}(b)$, $b/p \equiv$* $\texttt{(set! }n\ f\texttt{)}$, *$b \rightarrow_{N+1} e$, $e/p \equiv$* $\texttt{(set! }n\ g\texttt{)}$, *$\rho(b,p2) = 1$, and either (i) $a \in C_N$, or (ii) $a \in R_N$ and $b \in R_{N+1}$, then* $\texttt{(let ((}a\texttt{)) }b\texttt{)} \rightarrow_N \texttt{(let ((}g^-\texttt{)) }e_p[g]\texttt{)}$

19. *If $a \rightarrow_N d$, $p \in \Sigma_{N+1}(b)$, $b/p \equiv$* $\texttt{(lambda () }f\texttt{)}$, *$\nu(b,p) = m$, $b \rightarrow_{N+1} e$, $FV(f) \cap \{m+3, \cdots, m+2+N\} = \emptyset$, $\nu(e,p) = n$, $e/p \equiv$* $\texttt{(lambda () }g\texttt{)}$, *positions $p_1, \cdots, p_k$ are all the free occurrences in g satisfying $\rho(g,p_i) = n+2$, and either (i) $a \in C_N$, or (ii) $a \in R_N$ and $b \in R_{N+1}$, then*
$$\texttt{(let ((}a\texttt{)) }b\texttt{)} \rightarrow_N \texttt{(let ((}d\texttt{)) }e_p[\texttt{(lambda ()}g_{p_1,\cdots,p_k}[d\overbrace{^{+\cdots+}}^{n+2}]\texttt{)}]\texttt{)}.$$

20. *If $b \rightarrow_N e$ and $c \rightarrow_N f$, then*
$$\texttt{(while true }b\ c\texttt{)} \rightarrow_N \texttt{(let ((}x\ e\texttt{)) (while }f\ e\ f\texttt{))}.$$

21. $\texttt{(while false }b\ c\texttt{)} \rightarrow_N \texttt{nil}$.

We often omit the subscript N in $\rightarrow_N$. We call the rule 12 β-rule, and the rules 16, 17, 18 and 19 `let`-rules. As in $\lambda\sigma$-calculus, β-rule just adds a new environment to a term, and does not perform substitution. Later, `let`-rules will resolve this environment and perform the substitution. In `let`-rules, we may evaluate a subterm at a position in the set $\Sigma(a)$. Rules 17 and 19 do substitution for occurrences of the variable bound by this `let`. Rule 18 is the execution of assignment. Rule 16 eliminates `let` environment if there are no occurrences of the variable bound by this `let`.

Note that `let`-rules are extended from the original Λ! by the reason stated in Section 5.3.2.

Chapter 6

Conclusion

Summary of the thesis

We have studied the paradigm of Constructive Programming based on Sato's Reflective Proof Theory ($\mathcal{RPT}$), and also studied extensions of our framework. $\mathcal{RPT}$ is a type-free first-order theory for Constructive Programming. $\mathcal{RPT}$ extended Aczel's Frege structures in three directions; explicit proof-terms, the built-in reflection mechanism, and the inductive definition mechanism. These three extensions are quite important in developing the paradigm of Constructive Programming.

In Chapter 2, we proposed a formal system RPT for the semantical theory $\mathcal{RPT}$. Our formal system captures essential features of $\mathcal{RPT}$. We showed that many substantial theorems can be internally proved in RPT. In particular, we showed that the disjunction property and the term-existence property are expressed and proved in RPT. Since these properties are metatheorems in other theories such as first-order logic, these results showed the expressiveness of the reflection mechanism in RPT.

We also studied metamathematical properties of RPT. For logical systems, the strong normalization property is one of the most important properties. We showed that the weak normalization property does not hold for a naive formulation of RPT. We analyzed this failure, and proposed an appropriate restriction for the reduction relation for the proof-terms. We proved the strong normalization theorem for a subsystem of RPT, namely, RPT without inductive definitions and the equal-left rule. We then proved that the weak normalization property holds for RPT without inductive definitions. As a corollary, we obtained the consistency of each system.

In Chapter 3, we described an overview of our implementation of the Constructive Programming System based on RPT. Our system provides supports for men to develop proofs in RPT; it checks the correctness of the proof, and also it proves several kinds of theorems automatically. Moreover, our system extracts programs from proofs automatically, improves efficiency of the programs for many cases, and provides the execution environment of programs. One of the characteristic points of our system is that it is implemented by the programming language Λ, which is at the same time the object language of RPT.

We presented a mechanized proof of the Church-Rosser property of our calculus Λ. We also presented a concrete example of Constructive Programming taking the **append** program as an example. we first developed a proof of a given specification formula using the system, and then the system automatically extracted a program from the proof, and transformed it to a more efficient one. We described the theory which justifies the program transformation.

In Chapter 4, we studied a stronger reflection mechanism than RPT. We proposed the mechanism of *half-monotone* inductive definitions, which can be used to re-define the provability relation internally. We gave a theory and a realizability interpretation. As an application of this mechanism, we showed that a refinement of provability relation can be defined, and that we can formally state the relationship between an original and a refined provability relations. Moreover, we can extract an optimization program from this relationship.

In Chapter 5, we studied properties of our programming language Λ!. In order to extend our results to more realistic programs, our language must have the assignment and the `while` statements as in imperative programming languages. Sato designed Λ! as an extension of Λ by these statements. Λ! has a close relationship to the $\lambda\sigma$-calculus of the explicit substitution by Abadi et al. We studied some conservativeness results on Λ! over the $\lambda\sigma$-calculus and the pure λ-calculus.

Future Work

For future work, we have the following directions:

The first one is to extend RPT so that it can directly reason about the programming language Λ!. Then we can extract correct programs with the assignment and the `while` statements by our system.

The second one is to amalgamate the mechanism of half-monotone inductive definitions to RPT. We already interpreted the Logical Theory of Constructions, a formalized theory for Frege structures using our mechanism, therefore it should be possible to construct the theory RPT with the half-monotone definitions smoothly.

The third one is to extend our Constructive Programming System so that these extensions for RPT are reflected.

Also we should work on improvement of the proof-system itself. For the system to be more powerful, we should exploit automatic proof generation for RPT, and the user-interface of the system.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy, "Explicit Substitutions", *17th Annual ACM Symposium on Principles of Programming Languages*, pp. 31-46, 1990.

[2] P. Aczel, "Frege Structures and the Notions of Proposition, Truth and Set", *The Kleene Symposium* (Barwise, J., et al. eds.), North-Holland, pp. 31–59, 1980.

[3] P. Aczel, D. Carlisle, and N. Mendler, "Two frameworks of theories and their implementation in Isabelle", *Logical Frameworks* (G. Huet and G. Plotkin eds.), Cambridge University Press, pp. 3-39, 1991.

[4] S. Allen, "A non-type-theoretic definition of Martin-Löf's types", *Proc. 2nd Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 215-221, 1987.

[5] S. Allen, "A non-type-theoretic semantics for type-theoretic language", *Ph. D. Thesis, Cornell University*, 1987.

[6] S. Allen, R. L. Constable, D. Howe, and W. Aitken, "The Semantics of Reflected Proof", *Proc. 5th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 95-105, 1991.

[7] M. Beeson, *Foundations of Constructive Mathematics*, Springer-Verlag, 1985.

[8] E. Bishop, *Foundations of Constructive Analysis*, McGraw-Hill, 1967.

[9] R. S. Boyer and J. S. Moore, *A Computational Logic*, Academic Press, 1979.

[10] R. L. Constable, et al., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[11] T. Coquand and G. Huet, "The Calculus of Constructions", *Information and Computation*, Vol. 76, pp. 95–120, 1988.

[12] P.-L. Curien, "Categorical Combinators", *Information and Control* **69**, pp. 188-254, 1986.

[13] N. G. de Bruijn, "Lambda-calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation", *Indag. Mat.*, 34, pp. 381-392, 1972.

[14] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, and B. Werner, "The Coq Proof Assistant User's Guide, Version 5.8," Project Formel, INRIA-Rocquencourt, 1993.

[15] P. Dybjer, "Universes and a General Notion of Simultaneous Inductive-Recursive Definition in Type Theory", *Proc. of the 1992 Workshop on Types for Proofs and Programs* (B. Nordström et al eds.), Baastad, 1992.

[16] P. Dybjer, "A General Notion of Simultaneous Inductive-Recursive Definition in Type Theory", Draft, 1993.

[17] P. Dybjer, "Inductive Families", *Formal Aspects of Computing*, Vol 6, pp. 440-465, 1994.

[18] S. Feferman, "Constructive Theories of Functions and Classes", *Logic Colloquium '78* (Boffa, M., et al. eds.), North-Holland, pp. 159–224, 1979.

[19] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*, Cambridge, 1989.

[20] M. J. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science **78**, 1979.

[21] S. Hayashi and H. Nakano, **PX**: *a computational logic*, MIT Press, 1988.

[22] S. Hayashi and S. Kobayashi, *Foundations of Constructive Programming* (in Japanese, *Kouseiteki Puroguramingu no Kiso*) Yusei-sha, 1991.

[23] S. Hayashi, "Singleton, Union and Intersection Types for Program Extraction", *Information and Computation* Vol. 109, Nums 1 and 2, pp. 174-210, 1994.

[24] W. A. Howard, "The Formulae-as-types Notion of Constructions", in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp. 479-490. 1980.

[25] D. J. Howe, "Equality in Lazy Computation Systems", Proc. 4th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, pp. 198–203, 1989.

[26] Y. Kameyama and M. Sato, "Reflective Proof Theory and its Proof System" (in Japanese), Computer Software, JSSST, Vol. 12, No. 2, pp. 32-51, 1995.

[27] Y. Kameyama, "A Type-Free Theory of Half-Monotone Inductive Definitions", International Journal of Foundations of Computer Science, Vol. 6, No. 3, pp. 203-234. 1995.

[28] Z. Luo and R. Pollack, "LEGO Proof Development System: User's Manual", LFCS Technical Report ECS-LFCS-92-211, Edinburgh University, 1992.

[29] P. Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, 1984.

[30] B. Nordström, K. Petersson and J. Smith, *Programming in Martin-Löf's type theory*, Oxford, 1990.

[31] D. Park, "Finiteness is Mu-ineffable", *Theoretical Computer Science*, Vol 3, pp. 173-181, 1976.

[32] C. Paulin-Mohring, "Extracting F_ω's programs from proofs in the calculus of construction", *Proc. 16th Annual ACM Symposium in Principles of Programming Languages*, pp. 89-104, 1989.

[33] M. Sato and Y. Kameyama, "Constructive Programming in SST", Proc. the Japanese-Czechoslovak Seminar on Theoretical Foundations of Knowledge Information Processing, Prague, pp. 23–30, 1989.

[34] M. Sato, "Adding Proof Objects and Inductive Definition Mechanisms to Frege Structures", Proc. International Conference on Theoretical Aspects of Computer Software, Lecture Notes in Computer Science **526** (T. Ito and A. Meyer eds.), Springer, pp. 53–87, 1991.

[35] M. Sato and T. Sakurai, *Foundation of Theory of Programs* (*Puroguramu-no-Kisoriron*, in Japanese), Iwanami-Shoten, 1991.

[36] M. Sato, "A Purely Functional Language with Encapsulated Assignment", Proc. International Symposium TACS '94, Lecture Notes in Computer Science **789** (M. Hagiya and J. C. Mitchell eds.), pp. 179 -202, 1994.

[37] M. Sato and Y. Kameyama, "Conservativeness of Λ over $\lambda\sigma$-calculus", Logic, Language and Computation, Lecture Notes in Computer Science **792** (N. D. Jones, M. Hagiya, and M. Sato eds.), pp. 73-94, 1994.

[38] N. Shankar, "A Mechanical Proof of the Church-Rosser Theorem", Journal of Association for Computing Machinery, Vol. 35, No. 3 , pp. 475–522, 1988.

[39] B.C. Smith, "Reflection and Semantics in Lisp", *Proc. 11th Annual ACM Symposium on Principles of Programming Languages*, pp. 23-35, 1984.

[40] S. Smith, "Reflective Semantics of Constructive Type Theory (Preliminary Report)", Lecture Notes in Computer Science **613**, Springer, pp. 33-45, 1991.

[41] C. Svensson, "A Normalization Proof for Martin-Löf's Type Theory", Ph D. Dissertation, Dept. of Computer Science, University of Göteborg, 1990.

[42] M. Takahashi, "Parallel Reductions in λ-Calculus", *Journal of Symbolic Computation*, Vol. 7, pp. 113–123, 1989.

[43] Y. Takayama, "Extended Projection: a New Technique to Extract Efficient Programs from Constructive Proofs", *Proc. Conference on Functional Programming Languages and Computer Architecture* ACM Press, 1989.

[44] Y. Takayama, "Extended Projection Method for Proof Complier" (in Japanese) Computer Software, JSSST, Vol. 7, No. 4, pp. 19-38, 1990.

[45] M. Tatsuta, "Program Synthesis Using Realizability", *Theoretical Computer Science*, Vol. 90, pp. 309-353, 1991.

[46] M. Tatsuta, "Two Realizability Interpretations of Monotone Inductive Definitions", *International Journal of Foundations of Computer Science*, Vol. 5, No. 1, pp. 1-21, 1994.

[47] A. S. Troelstra and D. van Dalen, *Constructivism in Mathematics*, Vol. 1, 2, 1988.

List of Publications by the Author

Major Publications

1. M. Sato and Y. Kameyama, "Constructive Programming in $\mathcal{SST}$", Proceedings of the Japanese-Czechoslovak Seminar on Theoretical Foundations of Knowledge Information Processing, Prague, pp. 23–30, 1989.

2. M. Sato and Y. Kameyama, "Conservativeness of Λ over $\lambda\sigma$-calculus", Logic, Language and Computation, Lecture Notes in Computer Science **792**, (N. D. Jones, M. Hagiya, and M. Sato eds.) pp. 73-94, 1994.

3. A. Yamanaka, Y. Kameyama, and M. Sato, "Implementation of a Purely Functional Language Λ with Encapsulated Assignment" (in Japanese), Proceedings of Workshop on Functional Programming JSSST'94, Lecture Notes/Software Science Series **10**, (M. Takeichi ed.), Kindai-Kagaku-sha, pp. 201-216, 1994.

4. Y. Kameyama and M. Sato, "Reflective Proof Theory and its Proof System" (in Japanese), Computer Software, JSSST, Vol. 12, No. 2, pp. 32-51, 1995.

5. Y. Kameyama, "A Type-Free Theory of Half-Monotone Inductive Definitions", International Journal of Foundations of Computer Science, Vol. 6, No. 3, pp. 203-234, 1995.

Oral Presentations

1. Y. Kameyama, "Axiomatic System for Concurrent Logig Programming Languages", US-Japan Workshop of Logic on Programs, Hawaii, 1987.

2. M. Sato and Y. Kameyama, "Constructive Programming based on $\mathcal{SST}/\Lambda$", (in Japanese) IPSJ SIG on Foundation of Software, 31-6(1–10), 1989.

3. Y. Kameyama, "Formalizing Metamathematical Theorems based on Constructive Logic $\mathcal{RPT}$" (in Japanese), Proceedings of Annual Convention of IPSJ, Vol. 1, pp. 47-48, 1991.

4. Y. Kameyama, "Proof System of $\mathcal{RPT}$" (in Japanese), Annual SLACS Workshop, Sendai, 1991.

5. Y. Kameyama, "Traffic Analysis of JAIN network" (in Japanese), Proceedings of Symposium on Inter-connectivity of Academic Networks in Japan, pp. 19-28, March 1992.

6. Y. Kameyama, "Tohoku-INET: Current Status and Future Problem" (in Japanese), Proceedings of Workshop on Regional Networks, Computer Center, University of Tokyo, 1992.

7. Y. Kameyama, "A New Assignment Method of IP Addresses" (in Japanese), Proceedings of IP Meeting '92, Fujisawa, pp.32-34, 1992.

8. Y. Kameyama, "Constructive Programming System based on Reflective Proof Theory" (in Japanese), Functional Logic Programming Symposium, Tsukuba, 1993.

9. Y. Kameyama, "Inductive Definition with Negative Occurrences and its Application", Annual SLACS Workshop, Nara, 1993.

10. Y. Kameyama, "Program Optimization using an Extension of Simultaneous-Inductive Definition", Functional and Logic Programming Symposium, Susono, July, 1994.

11. Y. Kameyama, "Optimization of Extracted Programs in Constructive Programming" (in Japanese), 11th Conference Proceedings of JSSST, D4-4, pp. 177-180, 1994.

12. Y. Kameyama, M. Tatsuta, and M. Sato, "On Strong Normalizability of Catch/Throw Calculi" (in Japanese), JSSST Special Interest Group on Programming Theory, Keihanna-Plaza, 1995.

13. Y. Kameyama, and M. Sato, "The Strong Normalizability of an Intuitionistic Natural Deduction System with the Catch and the Throw Rules", Workshop on Constructive Programming, Kyoto University, 1996.

Other Publications

1. Y. Kameyama and M. Hirabaru, "Academic Inter-university Network: JAIN", (in Japanese) Operations Research, Vol. 37, No. 12, pp.599-602, 1992.

IPSJ: Information Processing Society, Japan
JSSST: Japan Society for Software Science and Technology
SLACS Workshop: Workshop on Symbolic Logic and Computer Science