<table>
<thead>
<tr>
<th>Title</th>
<th>Reflection Mechanism in Constructive Programming(全文)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Kameyama, Yukiyoshi</td>
</tr>
<tr>
<td>Citation</td>
<td>Kyoto University</td>
</tr>
<tr>
<td>Issue Date</td>
<td>1996-07-23</td>
</tr>
<tr>
<td>URL</td>
<td><a href="https://doi.org/10.11501/3117211">https://doi.org/10.11501/3117211</a></td>
</tr>
<tr>
<td>Type</td>
<td>Thesis or Dissertation</td>
</tr>
<tr>
<td>Textversion</td>
<td>author</td>
</tr>
</tbody>
</table>
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.

RPT, Reflective Proof Theory, is a constructive logic proposed by Sato[34]. Since the reflection mechanism is built-in in RPT, it can be a suitable basis for our study. In Chapter 2, we first propose a formal system of RPT. We then give several theorems in the formal system, which show the expressive power of the reflection mechanism of 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 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 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-
A language $\Lambda$ 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 $\Lambda!$ (pronounced "lambda bang") was proposed by Sato[36] as an extension of $\Lambda$ in which the assignment and the while statements are introduced. We give some conservativeness results on $\Lambda!$ in Chapter 5.

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

Acknowledgements

I would like to express my heartfelt thanks to Professor Masahiko Sato of Kyoto University for encouragements and continual supports. I would like to thank Professor Takayasu Ito, Professor Tadashi 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

1 Introduction ............................................. 1
   1.1 Backgrounds ........................................ 1
   1.2 Goal of this research .............................. 4
   1.3 Outline of the thesis .............................. 5

2 Formalizing \( \mathcal{RPT} \) ............................... 7
   2.1 Semantical Theory \( \mathcal{RPT} \) ................. 7
   2.2 Formal System \( \mathcal{RPT} \) ....................... 10
      2.2.1 The target of the formalization ............... 10
      2.2.2 Terms and their reduction rules ............... 12
      2.2.3 Judgement and inference rules ............... 15
      2.2.4 \( \mathcal{RPT} \) and \( \mathcal{RPT}_0 \) ............... 22
   2.3 Several Theorems in \( \mathcal{RPT} \) ................ 22
      2.3.1 S combinator .................................. 22
      2.3.2 Set and Russell's paradox ................... 23
      2.3.3 Use of reflective tower-1 ................... 25
      2.3.4 Use of reflective tower-2 ................... 25
   2.4 Strong Normalizability of \( \mathcal{RPT} \) ............... 26
      2.4.1 Strong Normalizability and Weak Normalizability 27
      2.4.2 Strong Normalizability of \( \mathcal{RPT} \) .......... 27
      2.4.3 Correspondence between Proof-figures and Proof-terms 28
      2.4.4 Formalizing \( \mathcal{RPT}_0 \) .................. 29
      2.4.5 Reducibility and its properties ............... 30
      2.4.6 Properties of \( \mathcal{RPT}_0 \) ............... 37
      2.4.7 Properties of \( \mathcal{RPT}_0 \) and \( \mathcal{RPT} \) ........... 38
   2.5 Conclusion ........................................ 38

3 Constructive Programming System based on \( \mathcal{RPT} \) ...... 41
   3.1 Introduction ........................................ 41
   3.2 Overview of the System .......................... 41
      3.2.1 Interaction with the system ................. 43
      3.2.2 Automatic Proof Generation .................. 45
   3.3 Mechanized Proof of the Church-Rosser Theorem .... 46
3.3.1 Proof Method ........................................ 47
3.4 Program Synthesis ...................................... 48
3.4.1 The specification of append and its proof .......... 49
3.4.2 Improvement of extracted programs ................. 50
3.4.3 Eliminating Redundancy by Program Transformation 50
3.5 Conclusion ............................................... 55

4 Half-monotone Inductive Definitions ................. 59
4.1 Introduction ........................................... 59
4.2 The basic theory ...................................... 62
4.3 Inductive Definitions ................................ 64
4.3.1 Monotone inductive definition ..................... 65
4.3.2 Refined order ...................................... 66
4.3.3 Half-monotone inductive definition ............... 67
4.3.4 Restricted version of half-monotone inductive definition 68
4.3.5 A model of HMID .................................. 69
4.3.6 Example of HMID ................................... 71
4.4 Realizability interpretation and its soundness for BT+HMID 72
4.4.1 Realizability Interpretation ........................ 73
4.4.2 Soundness of Realizability Interpretation .......... 74
4.4.3 Verification of the HMONO-Q condition ........... 77
4.5 Provability Relations and its Refinement ............ 78
4.5.1 Defining a Provability Relation ................... 78
4.5.2 Re-defining the Provability Relation ............ 80
4.6 Interpretation of Other Theories ...................... 82
4.6.1 The Logical Theory of Constructions ............... 82
4.6.2 Interpreting LTC by BT+HMID ........................ 84
4.6.3 Reformulating HMID 0 on LTC 0 .................... 86
4.6.4 Interpretation of Martin-Löf's type theory ........ 86
4.7 Conclusion ............................................ 86

5 Conservativeness of Λ! over λσ-calculus ............... 89
5.1 Introduction ........................................... 89
5.2 λσ-calculus with de Bruijn index ...................... 90
5.3 Λ! and plet-calculus .................................. 92
5.3.1 A Functional Programming Language Λ! ............ 92
5.3.2 Modification to Λ! .................................. 92
5.3.3 Definition of Λ! .................................... 93
5.4 Translation of λσ-calculus into the pure-fragment ..... 96
5.4.1 Translation of λσ-calculus into plet-calculus ....... 96
5.4.2 Translation from plet-calculus to Λ! ............... 97
5.4.3 Properties of the translation Φ .................... 98
5.5 Translation of the pure-fragment into λσ-calculus .... 103

6 Conclusion ................................................ 105
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\cite{18, 10, 11, 21, 33, 30, 22}.

A constructive logic is a logic used for reasoning in constructive mathematics\cite{8, 7}. It is not a single logic; rather, it can be any logic which allows the BHK (Brouwer-Heyting-Kolmogorov)-interpretation\cite{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 $\lor$) A proof of a disjunctive formula $A \lor B$ 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 $A \lor \neg A$ 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\cite{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 \( \Pi \forall x \in S. \Sigma y \in T. A(x, y) \) where \( \Pi \) and \( \Sigma \) 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 \( p : \Pi \forall x \in S. \Sigma y \in T. A(x, y) \), 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\cite{21}, Sato's SST\cite{33} for untyped logics based on the intuitionistic logic, and Martin-Löf's type theory\cite{29}, Coquand and Huet's Calculus of Constructions\cite{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 \( RPT \), Reflective Proof Theory\cite{34} is yet another constructive logic which is intended to be a basis for Constructive Programming. \( RPT \) 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 \( RPT \) as a mixture of the two ways. It is interesting to study Constructive Programming based on \( RPT \).

**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\cite{21}, NuPrl system\cite{10} in Cornell university, Coq system\cite{14} in INRIA, and Pollack's LEGO system\cite{28}. Each system has its own characteristics. One demerit of these 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:

\[
\text{Prov}_{\Pi}(. "A") \supset A
\]

where "\( A \)" represents a formalized representation (such as encoding by Gödel numbers) of \( A \), and the predicate \( \text{Prov}_{\Pi}(-) \) 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 \( \text{Prov}_{\Pi}(. "A") \), 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):

\[
\text{If } A \text{ is a formula, then } A \supset A \text{ is provable.}
\]

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:

\[
\text{From } \vdash \forall x. \exists y. A(x, y),
\]

\[
\text{we have } \vdash \forall x. A(x, f(x)) \text{ for some term } f
\]

where \( \vdash \) 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 \( RPT \), we can actually prove a formalized version of the principle above as follows:

\[
\vdash_{\Pi + 1} (\vdash \forall x. \exists y. A(x, y) \supset \exists f.(\vdash \forall x. A(x, f(x)))
\]
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 \( \text{Prov} A \) above. In this example, the predicate \( \text{Prov}_A \) 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 and al'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 programs of a property 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 naïve 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 meta-theory, 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) meta-theorem. 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 \( \Lambda \). \( \Lambda \) 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 \( \Lambda \), which is a purely functional language with the assignment and the while statements. We have to analyze the properties of the new \( \Lambda \) 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 \( \Lambda \), 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 \( \Lambda \). 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 $\Lambda$ 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 $\Lambda$ (called $\Lambda'$ in this thesis) is an extension of $\Lambda$ in which the assignment and the while statements are introduced by Sato[36]. We will give some conservativeness results on $\Lambda'$ 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 $\mathcal{RPT}$, a formal system for $\mathcal{RPT}$. (In the following we will write $\mathcal{RPT}$ 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 $\mathcal{RPT}$. We then give several theorems which demonstrate the expressive power of $\mathcal{RPT}$.

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

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 $\lambda$ calculus. Let $M$ be a domain of the model. By an appropriate encoding, we can assume that
constants \( \bot, \top, \forall, \exists, \text{ and } \exists \) are included in \( M \). These constants correspond to the logical symbols \( \bot, \top, \forall, \exists, \text{ and } \exists \).

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 \( \lambda \)-terms), and that of \( Q \) is the set of true propositions (represented as \( \lambda \)-terms).

- \( \bot \in P \).
- \( \bot \notin Q \).
- \( \forall ab \in P \) if and only if \( a \in P \) and \( b \in P \).
- \( \exists ab \in Q \) if and only if \( a \in Q \) and \( b \in Q \).
- \( \forall ab \in P \) if and only if \( a \in P \) and \( b \in P \).
- \( \exists ab \in Q \) if and only if either (i) \( a \in Q \) and \( b \in P \), or (ii) \( a \in Q \) and \( b \in Q \).
- \( \exists ab \in P \) if and only if \( a \in P \) and \( b \in Q \).
- \( \forall ab \in Q \) if and only if \( ab \in P \) for any term \( b \).
- \( \exists ab \in Q \) if and only if \( ab \in P \) for any term \( b \).
- \( \exists a \in P \) if and only if \( ab \in P \) for some term \( b \).
- \( \exists a \in Q \) 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. Acel used Frege structures to analyze Russell’s paradox[2].

Sato proposed \( RPT \) with the following extensions to Frege structures:

### Extension of the domain

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

### 2.1. Semantical Theory \( RPT \)

#### Explicit proof

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

\( RPT \) 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 \( j = \langle a \rangle \) and \( j \vdash a \), respectively. Here the suffix \( j \) 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 \( RPT \). The judgement \( j \vdash a \) in \( RPT \) 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, \( RPT \) introduces the level for each proposition, and regards a metaproposition of level \( j \) as a level \( j \) 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 \( A(t) \). \( \square \)

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

\[
a \vdash_{\text{RPT}} ((\forall x. f(x)) \supset (\exists z. f(z)))
\]

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).

\( RPT \) has a general mechanism of inductive definitions. Inductive definition is treated in more depth in Chapter 4.

Sato proposed \( RPT \) by extending Frege structures with the four extensions above.

\( RPT \) 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 \( RPT \). A formalization of \( RPT \) is described in the next section.

2.2 Formal System RPT

This section presents the system \( RPT \), which is a formalization of \( RPT \).

2.2.1 The target of the formalization

The semantical theory \( RPT \) was given in [34].

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

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

Compared with \( RPT \), we put the following restrictions to \( RPT \).

1. Restriction on the level of reflective tower

In \( RPT \), 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 \( \omega \) 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,

\[
t \vdash_{\omega} \forall i < \omega. \forall a.((i = a) \supset a \supset a)
\]

is a true judgement if we put \( t \equiv \lambda x.y.z.z \). This judgement formalizes "for each level less than \( \omega \), 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 (\( \omega \) or a larger ordinal), nor level-variables. We therefore decided to restrict the level to 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 \( RPT \), 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, \( X(a) \lor (b = 0 \supset X(a)) \) 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 \( (X(x) \supset X(x)) \land X(x) \). Then \( X(x) \) appears in the left handside 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 \( RPT \).

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 syntactically strictly positive cases. This restriction allows us to remove \( \forall \epsilon^{+} \) and \( \forall^{+} \) which were used to express the (semantic) strictly positiveness in \( RPT \).

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:

\[
a = \text{nil} \lor a = \text{true} \lor a = \text{false} \lor \text{pair}(a) = \text{true} \lor \text{fun}(a) = \text{true}
\]

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.
2.2.2 Terms and their reduction rules

Terms of RPT are defined as follows.

Definition 1 (Term)

\[ t := x \mid \text{nil} \mid \text{null}\!(t) \mid \text{true} \mid \text{true}\!(t) \mid \text{false} \mid \text{false}\!(t) \mid \langle t, t \rangle \mid \text{car}\!(t) \mid \text{cdr}\!(t) \mid \text{pair}\!(t) \mid \lambda x.t \mid t(t) \mid \text{fun}\!(t) \mid \mu t \mid \text{if } t \text{ then } t \text{ else } t \text{ fi} \]

where \( x \) is a metavariable for variables. □

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

\[
\mu \lambda f.\lambda x.\lambda y.\text{if null}\!(x) \text{ then } y \text{ else } (\text{car}\!(x), f(\text{cdr}\!(x))(y)) \text{ fi}
\]

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, \( \text{nil} \) and \( \text{null}\!(t) \) are of the same kind, while \( \text{nil} \) and \( \text{true}\!(t) \) are of different kinds. Terms \( \text{null}\!(t) \), \( \text{true}\!(t) \), \( \text{false}\!(t) \), \( \text{pair}\!(t) \), and \( \text{fun}\!(t) \) 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 \( a_{x_1, \ldots, x_n} [b_1, \ldots, b_n] \) is the result of simultaneous substitution of \( b_1, \ldots, b_n \) for \( x_1, \ldots, x_n \) in the term \( a \).

\( FV(a) \) is the set of free variables in \( a \).

We also use the following abbreviations:

\[
ab \triangleq a(b) \\
ab(b_1, \ldots, b_n) \triangleq a(b_1) \cdots (b_n) \\
() \triangleq \text{nil}
\]

2.2. FORMAL SYSTEM RPT

\[
\begin{align*}
(a) & \triangleq \langle a, \text{nil} \rangle \\
\langle a \ b \rangle & \triangleq \langle a, \langle b, \text{nil} \rangle \rangle \\
\langle a \ b \ c \rangle & \triangleq \langle a, \langle b, \langle c, \text{nil} \rangle \rangle \rangle \\
\lambda xy.a & \triangleq \lambda x.\lambda y.a \\
\lambda xyz.a & \triangleq \lambda x.\lambda y.\lambda z.a
\end{align*}
\]

We then define canonical terms and normal terms.

Definition 2 (Canonical term)

\[
c := \text{nil} \mid \text{true} \mid \text{false} \mid \langle t, t \rangle \mid \lambda x.t
\]

Definition 3 (Normal term)

\[
n := x \mid \text{nil} \mid \text{true} \mid \text{false} \mid \langle n, n \rangle \mid \lambda x.n
\]

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 \( \text{pair}\!(a) \), 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:

\[
\begin{align*}
0 & \triangleq \text{nil} \\
i + 1 & \triangleq \langle \text{nil}, i \rangle
\end{align*}
\]

Propositions and judgements are also terms via the following encoding:

\[
\begin{align*}
a = b & \triangleq \langle 1 \ a \ b \rangle \\
a[b] & \triangleq \langle 2 \ a \ b \rangle \\
l \triangleright_i b & \triangleq \langle 3 \ i \ b \rangle \\
a \triangleright_i b & \triangleq \langle 5 \ i \ a \ b \rangle \\
a & \downarrow & \triangleq \langle 6 \ a \rangle \\
\langle a \ \& \ b \rangle & \triangleq \langle 7 \ a \ b \rangle \\
\langle a \ \& \ b \rangle & \triangleq \langle 8 \ a \ b \rangle
\end{align*}
\]
\[
\begin{align*}
\text{Definition 4 (Evaluation of terms)} & \quad \text{For two terms } a \text{ and } b, \text{ we define } a \rightarrow b \text{ as in the table below.} \\
\text{Reduction Rule} & \quad \text{Condition} \\
\#(a) \rightarrow \text{true} & \quad \#(a) \text{ is a recognizer term,} \\
& \quad a \text{ is canonical, and } \#(a) \text{ and} \\
& \quad a \text{ are of the same kind} \\
\#(a) \rightarrow \text{false} & \quad \#(a) \text{ is a recognizer term,} \\
& \quad a \text{ is canonical, and } \#(a) \text{ and} \\
& \quad a \text{ are of different kinds} \\
car((a, b)) \rightarrow a & \quad \text{true} \\
cdr((a, b)) \rightarrow b & \quad \text{false} \\
(\lambda x.a)b \rightarrow a[b] & \quad \text{false} \\
\lambda x.(\lambda y.a)x \rightarrow \lambda y.a & x \not\in \text{fv}(a) \\
\mu a \rightarrow a(\mu a) & \\
\text{if true then } a \text{ else } b & \rightarrow a \\
\text{if false then } a \text{ else } b & \rightarrow b \\
C[b] \rightarrow C[c] & \quad b \rightarrow c
\end{align*}
\]

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

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

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

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

The evaluation in \(\Lambda \) 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 \(\Lambda \) to implement our Constructive Programming System. Hence, \(\Lambda \) 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:

\[
\vdash_i p \qquad a \vdash_i p
\]

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 \(a, b, c, p, q, s, t, \ldots \) for terms, \(i, j \) for natural numbers, \(J, J \) 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**

\[
\begin{align*}
\vdash_i p \quad \vdash_{i+1} p & \quad (i > j) \\
\vdash_i p \quad a \vdash_j p & \quad a \vdash_{i+1} p \quad (i > j) \\
\end{align*}
\]

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

\[
\frac{0 \vdash a = b}{a \approx b} \quad \text{(where } a \approx b) \]
\[
\frac{\lambda x.0 \vdash \neg(a = b)}{\text{(with the side condition below)}}
\]
\[
\frac{a \vdash b, d \vdash c}{a \vdash b, [d] \vdash c} \quad \frac{a, b \vdash d}{a, [e] \vdash b}
\]
\[
\frac{a \vdash b = c}{a \vdash b \vdash c, d = e}
\]
\[
\frac{a \vdash b = c}{0 \vdash a = 0}
\]

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.

\[
\frac{p \vdash (a, b) = (c, d)}{p \vdash a = c} \quad \frac{p \vdash (a, b) = (c, d)}{p \vdash b = d}
\]
\[
\frac{p \vdash a = \text{null}(a) = \text{true}}{p \vdash a = \text{null}(a) = \text{false}}
\]
\[
\frac{p \vdash a = \text{false}(a) = \text{true}}{p \vdash a = \text{false}(a) = \text{false}}
\]
\[
\frac{p \vdash \text{fun}(a) = \text{true}}{p \vdash \text{fun}(a) = \text{false}} \quad \text{(with the side condition below)}
\]
\[
\frac{p \vdash \#(a) = \text{true}}{p \vdash a = \text{false}}
\]

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

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

\[
D(a) \triangleq \begin{cases} 
\text{if } \text{null}(a) = \text{true} & \text{then } (\text{true},0) \\
\text{else if } \text{true}(a) = \text{true} & \text{then } c_1 \\
\text{else if } \text{false}(a) = \text{true} & \text{then } c_2 \\
\text{else if } \text{pair}(a) = \text{true} & \text{then } c_3 \\
\text{else } & \text{then } c_4 \\
\end{cases}
\]

\[
c_1 \triangleq (\text{false},(\text{true},0))
\]
\[
c_2 \triangleq (\text{false},(\text{false},(\text{true},0)))
\]
\[
c_3 \triangleq (\text{false},(\text{false},(\text{false},(\text{true},0))))
\]
\[
c_4 \triangleq (\text{false},(\text{false},(\text{false},(\text{false},(\text{false},0)))))
\]

Rules for propositions

Most propositions in RPT are the same as formulas in the usual first-order logic; we have \( \text{true}, \text{false}, a \vdash, a = b, \vdash; p, \) and \( a \vdash p \) as atomic propositions, and \( \land, \lor, \forall, \exists, \) and \( \forall \) as logical connectives. The different points are two points:
• The judgements $|\Gamma, p\rangle$ and $\Gamma \vdash p$ 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 $p \& q$ to be a proposition, $p$ and $q$ must be propositions.

On the other hand, in order for $p \lor q$ 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 2.1.2.

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

In the last rule, we used the following definitions:

$\Gamma \vdash p \lor q$  \hspace{1cm} $a \vdash p \land q$

$0 \vdash \text{pair?}(a) = \text{true}$

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

In the last rule, we used the following definitions:

$P \triangleq 0 \vdash \text{car}(a) = \text{true}$

$Q \triangleq 0 \vdash \text{car}(a) = \text{false}$

$D \triangleq (\text{if car}(a) \text{ then } \lambda x. b \text{ else } \lambda x. fi)(\text{cdr}(a))$
These rules are called the &-intro, the &-elim-0, the &-elim-1, the &-intro, the &-elim-0, the &-elim-1, and the &-elim-2 inference rules, respectively.

\[
\begin{align*}
[a &\vdash p] \\
& \vdash, p \\
& \vdash, q \\
& \lambda x.a \vdash, p \supset q (x \text{ is an eigen variable}) \\
a &\vdash, p \\
& a \vdash, q \\
& \lambda x.a \vdash, p \supset q \\
& a \vdash, p \supset q \\
& b \vdash, p
\end{align*}
\]

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.

\[
\begin{align*}
[a &\vdash p] \\
& \vdash, p \\
& \vdash, q \\
& b \vdash, p
\end{align*}
\]

Theses rules correspond to the fold and unfold operations.

\[
\begin{align*}
& \vdash, F \\
& \vdash, [a] \\
& b \vdash, F_{x,f}[a,P] \\
& b \vdash, P[a] \\
& b \vdash, F_{x,f}[a,P]
\end{align*}
\]

This rule represents the induction principle. Here \( \text{indp} \) 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 \( \forall x.\vdash, p(x) \) in RPT. In order to define a predicate for this proposition, we need to define \( \text{PF} \) as follows:

\[
\begin{align*}
& \vdash, p \\
& \forall x.\vdash, p(x) \\
& \vdash, p(x)
\end{align*}
\]

Since \( \text{PF} \) 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 \( \text{Nat} \), the predicate for natural numbers.

\[
\begin{align*}
& \vdash, \forall x.\vdash, p(x) \\
& \vdash, f \vdash, (\text{Nat}(x) \supset p(x))
\end{align*}
\]

where \( \text{suc}(y) \equiv (0, y) \). We can easily infer \( \text{Nat}[n] \) is true for each (encoded) natural number \( n \) defined before. The proof-term of the induction principle for \( \text{Nat} \) is:
CHAPTER 2. FORMALIZING RPT

2.2.4 RPT and RPT

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

Theorem 1 (Soundness) RPT (after modification) is a model of RPT. □

Corollary 1 (Consistency) RPT is consistent. □

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 \( 0 \vdash a = b \) 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 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:

\[
F \triangleq (A \supset B \supset C) \supset (A \supset B) \supset A \supset C
\]

In this definition we used abbreviations in Lisp such as \( \text{cadr} \). It is easily shown that the induction principle for \( \text{Nat} \) is equivalent to the usual mathematical induction.

2.3.2 Set and Russell's paradox

In 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-1 proposition. Then \( p \) generates the level-1 set \( \{ x \mid p(x) \text{ is true} \} \) by comprehension.

In RPT, this fact can be internally formalized again. Let \( \text{Set} \) and \( \epsilon_i \) be the following:

\[
\text{Set}[p] \triangleq \forall x. (\neg\epsilon_i p(x))
\]

\[
\epsilon_i p \triangleq \text{Set}[p] \& p(a)
\]

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:

\[
P_1 \equiv \frac{x \vdash_0 A \supset B \supset C \quad y \vdash_0 A \supset B \quad z \vdash_0 A}{xz \vdash_0 B} \quad \frac{x \vdash_0 A \supset B \supset C \quad y \vdash_0 A \supset B \quad z \vdash_0 A}{yz \vdash_0 C}
\]

\[
P_2 \equiv \frac{x \vdash_0 A \supset B \quad \lambda y. z(x(y)) \vdash_0 A \supset C}{\lambda y. z(x(y)) \vdash_0 A \supset C}
\]

\[
P_3 \equiv \frac{\vdash_0 C}{\vdash_0 B} \quad \frac{\vdash_0 A \supset (\vdash_0 B) \supset (\vdash_0 C) \supset (S \vdash_0 F)}{S \vdash_0 F}
\]

where \( T = (\vdash_0 A) \supset (\vdash_0 B) \supset (\vdash_0 C) \supset (S \vdash_0 F) \).

The proof \( P_3 \) is a formalized version of the metatheorem.
The proposition \( \text{Set}_i[p] \) 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 \( \text{Set}_i[p] \wedge p(a) \), then \( a \in_i p \) is not necessarily a proposition.

As an example of sets, the set of natural numbers, \( \text{NatSet} \) is defined as follows:

\[
\text{NatSet} \triangleq \lambda x. \text{Nat}[x]
\]

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

Russell’s paradox in the naive set theory is that, if we can define the set \( R \) as \( R \triangleq \{ x | x \notin x \} \), then we have inconsistency. The modern set theory, namely, \( \text{ZF} \) set theory avoids this paradox by excluding the set comprehension rule; then we cannot define \( R \) as a set in \( \text{ZF} \) set theory. In \( \text{RPT} \), 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.

\[
\begin{align*}
\lambda u.a(\text{car}(u)) & \vdash \neg(R \in_0 R) \\
\lambda u.0 & \vdash_2 \text{Set}_1[R] \\
\langle \lambda u.0, \lambda w.a(\text{car}(w)) \rangle & \vdash_2 R \in_1 R
\end{align*}
\]

**Proof.** Define \( a \) as

\[
\lambda pqyz.\text{if cdr}(z) \text{then } (\text{true},(0,\text{cdr}(z))) \text{else } (\text{false},(0,\text{cdr}(z)))
\]

Then we can prove the theorem easily. \( \square \)

**Theorem 3** The disjunction property “if \( p \lor q \) is provable, then \( p \) or \( q \) is provable.” is formalized and proved in \( \text{RPT} \). Namely, we can prove the following judgement

\[
a \vdash_{i+1} \forall p. \forall q. \langle \forall i, p \lor q \rangle \supset (\forall i, p) \lor (\forall i, q)
\]

for some term \( a \).

**Proof.** Define \( a \) as \( \lambda pqyz.\text{if cdr}(z) \text{then } (\text{true},(0,\text{cdr}(z))) \text{else } (\text{false},(0,\text{cdr}(z))) \rangle \text{fi} \).

Then we can prove the theorem easily. \( \square \)

**Theorem 4** The term existence property “if \( \exists x. p \) is provable, then we can effectively find a term \( t \) and the proof of \( p_2[t] \)” is formalized and proved in \( \text{RPT} \). Namely, we can prove the following judgement

\[
a \vdash_{i+1} \forall f. \langle \text{PF}[f] \supset (\forall i, \exists x. f(x)) \supset (\exists z, \forall i, f(z)) \rangle
\]

for some term \( a \).

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

**Proof.** Define \( a \) as \( \lambda pqyz.\text{if cdr}(z) \text{then } (\text{true},(0,\text{cdr}(z))) \text{else } (\text{false},(0,\text{cdr}(z))) \rangle \text{fi} \).

Then we can prove the theorem easily. \( \square \)

**Remark 1** The term \( a \) is a closed proof-term in \( \text{RPT} \) which cannot be normalized. In other words, the proofs (proof-terms or equivalently proof-figures) in \( \text{RPT} \) 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 \( \text{RPT} \) is that we can internally express meta-theorems 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 metatheoretical arguments. For \( \text{RPT} \), we can prove the internally formalized version of these properties.

In the following we abbreviate the proposition \( \forall x. (x \vdash_i p) \) as \( \vdash_i p \).

**Theorem 3** The disjunction property “if \( p \lor q \) is provable, then \( p \) or \( q \) is provable.” is formalized and proved in \( \text{RPT} \). Namely, we can prove the following judgement

\[
a \vdash_{i+1} \forall p. \forall q. \langle \forall i, p \lor q \rangle \supset (\forall i, p) \lor (\forall i, q)
\]

for some term \( a \).

**Proof.** Define \( a \) as \( \lambda pqyz.\text{if cdr}(z) \text{then } (\text{true},(0,\text{cdr}(z))) \text{else } (\text{false},(0,\text{cdr}(z))) \rangle \text{fi} \).

Then we can prove the theorem easily. \( \square \)

**Theorem 4** The term existence property “if \( \exists x. p \) is provable, then we can effectively find a term \( t \) and the proof of \( p_2[t] \)” is formalized and proved in \( \text{RPT} \). Namely, we can prove the following judgement

\[
a \vdash_{i+1} \forall f. \langle \text{PF}[f] \supset (\forall i, \exists x. f(x)) \supset (\exists z, \forall i, f(z)) \rangle
\]

for some term \( a \).

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

**Proof.** Define \( a \) as \( \lambda pqyz.\text{if cdr}(z) \text{then } (\text{true},(0,\text{cdr}(z))) \text{else } (\text{false},(0,\text{cdr}(z))) \rangle \text{fi} \).

Then we can prove the theorem easily. \( \square \)

### 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 \( a \vdash P \) mean the term \( a \) is a realizer of the formula \( P \). Then \( a \vdash P \) is semantically similar to the judgement \( a \vdash_i P \) in \( \text{RPT} \).

Harrop formulas are defined as below in the first-order logic:
Definition 6 (Harrop formula in the first-order logic)

\[ H ::= \text{true} \mid \text{false} \mid t = t \mid H \land H \]
\[ \mid P \supset H \mid \forall x. H \]

where \( P \) is an arbitrary formula. \( \Box \)

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 \( \text{FV}(H) \subseteq \{x\} \) holds. Then we have \( H \supset (t(x) \lor H) \) for some term \( t \). \( \Box \)

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)

\[ \text{HPP}_i \triangleq \lambda f. \lambda p. p = \lambda x. \text{true} \]
\[ \lor p = \lambda x. \text{false} \]
\[ \lor \exists y. \exists z. (p = (\lambda x. y(x) = z(x))) \]
\[ \lor \exists y. \exists z. (p = (\lambda x. q(x) \land r(x)) \land f[q] \land f[r]) \]
\[ \lor \exists y. \exists z. (p = (\lambda x. q(x) \lor r(x)) \land \forall x. (\exists y. q(x) \land f[r])) \]
\[ \lor \exists y. (p = \lambda x. y(x) \land f[q]) \]

Theorem 6 We can prove the following for some term \( a \).

\[ a \vdash_{i+2} \forall p. (\text{HPP}_i[p] \supset \forall x. (\exists y. q(x)) p(x))) \]

Theorem 7 We can prove the following for some term \( a \).

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

These two theorems are proved by the induction on the predicate \( \text{HPP}_i \). 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 \( \text{HPP}_i \).

2.4 Strong Normalizability of RPT

In this section we study several metamathematical properties of the formal system RPT and its subsystem \( \text{RPT}_0 \). \( \text{RPT}_0 \) 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,

\[ \Pi \]
\[ a \vdash_{i+1} \Phi \]
\[ \{ (a, b) \vdash_{i+1} A \land B \} \] \text{\land-intro} \]
\[ \{ (a, b) \vdash_{i+1} A \land B \} \] \text{\land-elim} \]
\[ \text{car}((a, b)) \vdash_{i+1} A \]

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

\[ \Pi \]
\[ a \vdash_{i+1} \]

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

\[ \text{car}((a, b)) \rightarrow a \]

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 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 \( \bot \) (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 \( \lambda \)-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:

\[
\frac{a_j \vdash j \ p \quad c \vdash j \ d = e}{a_j [c] \vdash j \ b}
\]

By this rule, we can replace a proof-term to an equal term. It follows that, even if \(a_j [d]\) is strongly normalizing, the resulting proof-term \(a_j [e]\) 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.

\[
\frac{a \vdash j \ p \quad (i > j)}{a \vdash_j (a \vdash_j p)} \quad \frac{a \vdash_j (b \vdash_j p) \quad (i > j)}{a \vdash_j p}
\]

2.4.4 Formalizing RPT

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)**

\[
t ::= \text{up}(t) \mid \text{down}(t) \mid \text{freeze}(t)
\]

**Definition 9 (Neutral Term)** A term \(a\) is called neutral if it is of the form \(\text{car}(b)\), \(\text{cdr}(b)\), \(b(c)\), or \(\text{down}(b)\), 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:

\[
\text{down}(\text{up}(a)) \rightarrow a \\
\text{freeze}(a) \rightarrow a
\]
CHAPTER 2. FORMALIZING RPT

Note that, the relations \( \rightarrow, \rightarrow^*, \text{ and } \approx \) in this Section are those extended by these rules.

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

**Definition 11 (Restricted Reduction)** \( a \rightarrow b \) 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 \( \rightarrow \) is not Church-Rosser, since

\[
(\lambda x.\lambda y.x)(\texttt{car}((0,0))) \rightarrow \lambda y.\texttt{car}((0,0)),
\]

and

\[
(\lambda x.\lambda y.x)(\texttt{car}((0,0))) \rightarrow (\lambda x.\lambda y.x)(0) \rightarrow \lambda y.0.
\]

The inference rules of \( \text{RPT}_0 \) 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.

\[
p \vdash_i \text{up}(a) = \text{up}(b) \\
p \vdash_i a = b
\]

The modification of the four rules are as follows:

\[
\begin{array}{c}
\frac{a \vdash_j p}{\text{up}(a) \vdash_i (a \vdash_j p)} (i > j) \\
\frac{a \vdash_j (k \vdash_j p)}{\text{down}(a) \vdash_i p} (i > j) \\
\frac{a \vdash_i \forall p}{a(\text{freeze}(t)) \vdash_i p(t)} \\
\frac{a(\text{freeze}(t)), a \vdash_i \exists p}{a(\text{freeze}(t), a) \vdash_i \exists p}
\end{array}
\]

The modification for these four rules are mainly introducing the function symbols \text{up}, \text{down}, and \text{Freeze}.

The system \( \text{RPT}_0 \) is \( \text{RPT}_0 \) without the left-equal rule.

In the following, we shall prove the strong normalizability for \( \text{RPT}_0 \), and then obtain some proof-theoretic results for \( \text{RPT}_0 \) and \( \text{RPT} \).

### 2.4.5 Reducibility and its properties

This subsection presents the main theorem of this section, from which we have the SN property for \( \text{RPT}_0 \).

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

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

- \( A \) is true,
  - \( \text{Red}_i(A) \) is defined, and is equal to \( S \) where \( S \) is the set of closed terms which strongly normalize to \( 0 \).
- \( A \) is false,
  - \( \text{Red}_i(A) \) is defined, and is equal to \( \{\} \).
- \( A \) is \( B \land C \),
  - \( \text{Red}_i(A) \) is defined if and only if both \( \text{Red}_i(B) \) and \( \text{Red}_i(C) \) are defined.
- \( A \) is \( B \lor C \),
  - \( \text{Red}_i(A) \) is defined if and only if both \( \text{Red}_i(B) \) and \( \text{Red}_i(C) \) are defined.
- \( A \) is \( B \rightarrow C \),
  - \( \text{Red}_i(A) \) is defined if and only if \( \text{Red}_i(B) \) is defined, and either \( \text{Red}_i(B) \) is empty, or \( \text{Red}_i(C) \) is defined.
- \( A \) is \( B \& C \),
  - \( \text{Red}_i(A) \) is defined if and only if \( \text{Red}_i(B) \) is defined, and either \( \text{Red}_i(B) \) is empty, or \( \text{Red}_i(C) \) is defined.
- \( A \) is \( B \lor C \),
  - \( \text{Red}_i(A) \) is defined if and only if \( \text{Red}_i(B) \) is defined, and either \( \text{Red}_i(B) \) is empty, or \( \text{Red}_i(C) \) is defined.
- \( A \) is \( \forall B \),
  - \( \text{Red}_i(A) \) is defined if and only if for every closed term \( b \), \( \text{Red}_i(B(b)) \) is defined.
\* A is \(\exists B\), 
\[ \text{Red}_{i}(A) \text{ is defined if and only if for every closed term } b, \text{Red}_{i}(B(b)) \text{ is defined.} \]
\[ a \in \text{Red}_{i}(A) \text{ holds if and only if there exist } c \text{ and } d \text{ such that } a \approx (c, d), \text{ and } \text{cdr}(a) \in \text{Red}_{i}(B(\text{car}(a))). \]
\* A is \(\vdash_{j} B\), 
\[ \text{Red}_{i}(A) \text{ is defined if and only if } j < i. \]
\[ \text{Red}_{i}(A) \text{ is equal to } S \text{ if } \text{Red}_{i}(B) \text{ is defined, and } \emptyset \text{ otherwise.} \]
\* A is \(\vdash_{j} B\), 
\[ \text{Red}_{i}(A) \text{ is defined if and only if } j < i. \]
\[ a \in \text{Red}_{i}(A) \text{ holds if and only if } \text{Red}_{i}(B) \text{ is defined, down}(a) \in \text{Red}_{i}(B), \text{ and there exists a term } c \text{ such that } b \equiv c, \text{ and } c \in \text{Red}_{i}(B). \]
\* if \(A \equiv B\) and \(\text{Red}_{i}(B)\) has been defined, then \(\text{Red}_{i}(A)\) is equal to \(\text{Red}_{i}(B)\). 
\[
\text{This finished the definition of } \text{Red}_{i}(A). 
\]

This inductive definition is well-defined, since the set of terms \(A\) for which \(\text{Red}_{i}(A)\) 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 \(a \in \text{Red}_{i}(A)\) then \(a\) is strongly normalizing.

(CR2) if \(a \in \text{Red}_{i}(A)\), and \(a \rightarrow_{1} a'\) then \(a' \in \text{Red}_{i}(A)\).

(CR3) if \(a\) is neutral, closed, and for every \(a'\) such that \(a \rightarrow_{1} a'\), \(a' \in \text{Red}_{i}(A)\), then \(a \in \text{Red}_{i}(A)\).

Here, \(a \rightarrow_{1} a'\) is the one-step reduction for \(\rightarrow_{1}\). We refer to these properties as (CR).

**Lemma 2** For each natural number \(i\) and a proposition \(A\), the set \(\text{Red}_{i}(A)\) 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 \(\text{Red}_{i}(A)\).

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 \(\text{Red}_{i}\). For example, \(a \in \text{Red}_{i}(B \land C)\) must be in the form of \((c, d)\) for some \(c\) and \(d\). In proving (CR2) and (CR3), it is automatically guaranteed by virtue of the modified definition of neutrality.

\* A is true,

Trivial.

2.4. STRONG NORMALIZABILITY OF RPT

\* A is false,

There is no \(a \in \text{Red}_{i}(A)\), so CR trivially holds.

\* A is \(b = c\),

If \(b \equiv c\), then \(a \in \text{Red}_{i}(A)\) means \(a\) is strongly normalizing to \(0\), so CR holds.

Otherwise, \(\text{Red}_{i}(A)\) is empty, so CR holds.

\* A is \(B \land C\),

1. Suppose \(a \in \text{Red}_{i}(B \land C)\). We have \(\text{car}(a) \in \text{Red}_{i}(B)\), and by the induction hypothesis, \(\text{car}(a)\) is strongly normalizing, so \(a\).

2. Suppose \(a \in \text{Red}_{i}(B \land C)\), and \(a \rightarrow_{1} a'\). Since \(\text{car}(a) \in \text{Red}_{i}(B)\) and \(\text{car}(a) \rightarrow_{1} \text{car}(a')\), \(\text{car}(a') \in \text{Red}_{i}(B)\). Similarly, \(\text{cdr}(a') \in \text{Red}_{i}(C)\), and then \(a' \in \text{Red}_{i}(B \land C)\).

3. Suppose \(a\) is neutral, and for every \(a'\) such that \(a \rightarrow_{1} a'\), \(a' \in \text{Red}_{i}(B \land C)\). Since \(a\) is neutral, every 1-step-reduct of \(\text{car}(a)\) is of the form \(\text{car}(a')\) where \(a \rightarrow_{1} a'\). Since \(\text{car}(a') \in \text{Red}_{i}(B)\), we have \(\text{car}(a) \in \text{Red}_{i}(B)\) by the induction hypothesis. Similarly we have \(\text{cdr}(b) \in \text{Red}_{i}(C)\) and then \(a \in \text{Red}_{i}(B \land C)\).

\* A is \(B \lor C\),

1. Suppose \(a \in \text{Red}_{i}(B \lor C)\). We have \(\text{cdr}(a) \in \text{Red}_{i}(B)\), or \(\text{cdr}(a) \in \text{Red}_{i}(C)\). By the induction hypothesis, \(\text{cdr}(a)\) is strongly normalizing in either case, so is \(a\).

2. Suppose \(a \in \text{Red}_{i}(B \lor C)\), and \(a \rightarrow_{1} a'\). Assume \(\text{car}(a) \approx \text{true}\). Since \(\text{cdr}(a) \in \text{Red}_{i}(B)\) and \(\text{cdr}(a) \rightarrow_{1} \text{cdr}(a')\), we have \(\text{cdr}(a') \in \text{Red}_{i}(B)\). We also have \(\text{car}(a') \approx \text{true}\), so we have \(a' \in \text{Red}_{i}(B \lor C)\). Similarly for the case of \(\text{car}(a) \approx \text{true}\).

3. Suppose \(a\) is neutral, and for every \(a'\) such that \(a \rightarrow_{1} a'\), \(a' \in \text{Red}_{i}(B \lor C)\). Fix such an \(a'\). Then, either \(\text{car}(a') \approx \text{true}\) and \(\text{cdr}(a') \in \text{Red}_{i}(B)\), or \(\text{car}(a') \approx \text{false}\) and \(\text{cdr}(a') \in \text{Red}_{i}(C)\). Assume the former is the case. Since \(a\) is neutral, every 1-step-reduct of \(\text{cdr}(a)\) is of the form \(\text{cdr}(a')\) where \(a \rightarrow_{1} a'\). Since \(\text{cdr}(a') \in \text{Red}_{i}(B)\), we have \(\text{cdr}(a) \in \text{Red}_{i}(B)\) by the induction hypothesis. We also have \(\text{car}(a) \approx \text{true}\), so \(a \in \text{Red}_{i}(B \lor C)\). Similarly for the latter case.

\* A is \(B \supset C\),

1. Trivial.

2. Suppose \(a \in \text{Red}_{i}(B \supset C)\), and \(a \rightarrow_{1} a'\). For every \(b \in \text{Red}_{i}(B)\), \(a(b) \in \text{Red}_{i}(C)\) holds. By the induction hypothesis, \(a'(b) \in \text{Red}_{i}(C)\) holds for all such \(b\). Also \(a'\) is clearly strongly normalizing, so \(a' \in \text{Red}(B \supset C)\).
3. Suppose \( a \) is neutral, and for every \( a' \) such that \( a \rightarrow_{1} a' \), we have \( a' \in \text{Red}_{i}(B \supset C) \). Fix such an \( a' \). Since \( a' \) is strongly normalizing, so is \( a \). For every \( b \in \text{Red}_{i}(B) \), we have \( a'(b) \in \text{Red}_{i}(C) \). Since \( a \) is neutral, \( a(b) \) reduces to a term of the form \( a'(b') \) or \( a(b') \) where \( b' \rightarrow_{1} b' \). By the induction hypothesis, \( b' \) is strongly normalizing, so we eventually get \( a'(b') \in \text{Red}_{i}(C) \) by the induction on the length of the normalizing sequence of \( b \). Finally, we have \( a \in \text{Red}_{i}(B \supset C) \).

- \( A \) is \( B \& C \),
  Similar to the case of \( \wedge \).

- \( A \) is \( \forall B \),
  1. Suppose \( a \in \text{Red}_{i}(\forall B) \). Then, \( a(\text{freeze}(0)) \in \text{Red}_{i}(B(0)) \), hence \( a(\text{freeze}(0)) \) is strongly normalizing, so is \( a \).
  2. Suppose \( a \in \text{Red}_{i}(\forall B) \), and \( a \rightarrow_{1} a' \). Since \( a(\text{freeze}(b)) \rightarrow_{1} a'(\text{freeze}(b)) \), \( a'(\text{freeze}(b)) \in \text{Red}_{i}(B(b)) \) for every closed term \( b \). So \( a' \in \text{Red}_{i}(\forall B) \).
  3. Suppose \( a \) is neutral, and for every \( a' \) such that \( a \rightarrow_{1} a' \), we have \( a' \in \text{Red}_{i}(\forall B) \). So, \( a(\text{freeze}(b)) \in \text{Red}_{i}(B(b)) \) for any closed term \( b \). Since \( a \) is neutral, \( a(\text{freeze}(b)) \) reduces to the form \( a'(\text{freeze}(b)) \), so \( a(\text{freeze}(b)) \in \text{Red}_{i}(B(b)) \). Hence, we have \( a \in \text{Red}_{i}(\forall B) \).

- \( A \) is \( \exists B \),
  1. Suppose \( a \in \text{Red}_{i}(\exists B) \). We have \( \text{cdr}(a) \in \text{Red}_{i}(B(\text{car}(a))) \), and by the induction hypothesis, \( \text{cdr}(a) \) is strongly normalizing, so is \( a \).
  2. Suppose \( a \in \text{Red}_{i}(\exists B) \), and \( a \rightarrow_{1} a' \). Since \( \text{cdr}(a) \rightarrow_{1} \text{cdr}(a') \), \( \text{cdr}(a') \in \text{Red}_{i}(B(\text{car}(a'))) \) so we have \( \text{cdr}(a') \in \text{Red}_{i}(B(\text{car}(a'))) \), hence \( a' \in \text{Red}_{i}(\exists B) \).
  3. Suppose \( a \) is neutral, and for every \( a' \) such that \( a \rightarrow_{1} a' \), we have \( a' \in \text{Red}_{i}(\exists B) \). Since \( a \) is neutral, every 1-step-reduct of \( \text{cdr}(a) \) is of the form \( \text{cdr}(a') \) where \( a \rightarrow_{1} a' \). Since \( \text{cdr}(a') \in \text{Red}_{i}(B(\text{car}(a'))) \), we have \( \text{cdr}(a) \in \text{Red}_{i}(B(\text{car}(a'))) \). Then we have \( a \in \text{Red}_{i}(\exists B) \).

- \( A \) is \( \exists B \rightarrow B \),
  Similarly to the case of \( a = b \).

- \( A \) is \( B \rightarrow_{1} B \),
  1. Suppose \( a \in \text{Red}_{i}(B \rightarrow_{1} B) \). Then, \( \text{down}(a) \in \text{Red}_{i}(B) \), so by the induction hypothesis, \( a \) is strongly normalizing.
  2. Suppose \( a \in \text{Red}_{i}(B \rightarrow_{1} B) \), and \( a \rightarrow_{1} a' \). Then \( \text{down}(a) \in \text{Red}_{i}(B) \), and by the induction hypothesis, \( \text{down}(a') \in \text{Red}_{i}(B) \). Hence, \( a' \in \text{Red}_{i}(B \rightarrow_{1} B) \).
  3. Suppose \( a \) is neutral, and for every \( a' \) such that \( a \rightarrow_{1} a' \), \( a' \in \text{Red}_{i}(B \rightarrow_{1} B) \). Then, \( \text{down}(a') \in \text{Red}_{i}(B) \). Since \( a \) is neutral, every 1-step-reduct of \( \text{down}(a) \) is of the form \( \text{down}(a') \), and by the induction hypothesis, \( \text{down}(a) \in \text{Red}_{i}(B) \).

The following theorem is the main theorem of this chapter.

**Theorem 2** If we have a proof of the judgement \( J \) with assumptions \( J_{1}, \ldots, J_{n} \) in \( \text{RPT}_{5} \), then, we have that \( \text{Trans}(J_{1}, \theta), \ldots, \text{Trans}(J_{n}, \theta) \) imply \( \text{Trans}(J, \theta) \) where \( \theta \) 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 \( \text{len}(a) \) 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 handled, 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 \( \text{Red}_{i}(A) \) is in the form of the elimination style.

- **Case (prop-intro).**
  Since \( \text{Red}_{i}(p) \) is defined, and \( 0 \in S \), we have the conclusion.
- **Case (level-up).**
  By the reduction \( \text{down}(\text{up}(a)) \rightarrow a \), we have the conclusion.
- **Case (level-down).**
  This rule is essentially an elimination rule, so it is straightforward.
- **Case (\( \wedge \)-intro).**
  Suppose \( a \in \text{Red}_{i}(p) \), and \( b \in \text{Red}_{i}(q) \) for closed terms \( a, b, p, q \) and an ordinal number \( j \) which is less than \( a \).
  Consider the term \( \text{car}(\langle a, b \rangle) \). \( a \) and \( b \) are strongly normalizing. By the induction on \( \text{len}(a) + \text{len}(b) \), we have \( \langle a, b \rangle \in \text{Red}_{i}(p \& q) \).

2.4. STRONG NORMALIZABILITY OF RPT
2.4. STRONG NORMALIZABILITY OF RPT

We will show \((\lambda x. a)(\text{freeze}(b))\) \(\in \text{Red}_1((\lambda x. \Phi)\Phi)\) for any ground \(\Phi\). The key case is \(a \approx_1 \text{freeze}(b)\). By taking \(\theta\) as \(\Phi\) with \(x := \text{freeze}(b)\), we have the result.

Case 3 (\&-intro). Suppose \(a \theta \in \text{Red}_1(p, q)\) for any ground \(\theta\).

We will show \(\text{cdr}((\text{freeze}(t), a))\) \(\in \text{Red}_1((\lambda x. p)\text{car}(\text{freeze}(t), a)\Phi)\) for any ground \(\Phi\) by the induction on \(\text{len}(a)\). Note that \(\text{Red}_1((\lambda x. p)\text{car}(\text{freeze}(t), a)\Phi)\) is equal to \(\text{Red}_1(p, q)\).

The term \(\text{cdr}((\text{freeze}(t), a))\) can be reduced to either \(a\theta\) or \(\text{cdr}((\text{freeze}(t), a'))\) \(\theta\) where \(a\theta \rightarrow a\theta\). In the first case, we have the result by setting \(\theta\) in the assumption be \(\Phi\). In the second case, we also have the conclusion by using the induction hypothesis.

This finishes the proof. \(\Box\)

2.4.6 Properties of RPT

Theorem 3 If \(a \vdash b\) is proved in \(\text{RPT}_0\) without assumptions, \(a\) is strongly normalizing in the sense of \(\rightarrow\), or equivalently, a proof figure of \(a \vdash b\) is strongly normalizing.

Proof.

From the last theorem, we have \(\theta \in \text{Red}_1(\emptyset)\) for any ground substitution \(\theta\). Hence \(\theta\) is strongly normalizing (in the sense of \(\rightarrow\)) for any such \(\theta\). It follows that \(a\) is strongly normalizing. \(\Box\)

Theorem 4 \(\text{RPT}_0\) is consistent.

Proof.

By the theorem above, if we had a proof of \(a \vdash \text{false}\), then \(\text{Red}_1(\text{false})\) would have an element, but it contradicts the definition of \(\text{Red}_1(\text{false})\).

Corollary 3 (Subformula property) If \(a \vdash b\) is proved in \(\text{RPT}_0\) without assumptions, all the propositions in its proof figure are sub-propositions of \(b\).

Our definition of \(\text{Red}_1\) 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 \(a \vdash b \land c\), \(a \vdash b \lor c\), or \(a \vdash 3b\) is proved in \(\text{RPT}_0\) without assumptions, then \(a\) is a pair, namely, is equal to a term of the form \((b, c)\).
2. if \(a \vdash b \lor c\), then \(a\) is a pair, and its car-part is equal to either true or false.
3. if \(a \vdash b \lor c\), or \(a \vdash 3b\) is proved in \(\text{RPT}_0\) without assumptions, then \(a\) is a function, namely, is equal to a term of the form \(\lambda x.c\).

Corollary 4 (Disjunction Property) If \(a \vdash b \lor c\) is proved in \(\text{RPT}_0\) without assumptions, then we have either \(\text{cdr}(a) \vdash b\) or \(\text{cdr}(a) \vdash c\) in \(\text{RPT}_0\).
2.4.7 Properties of RPT₀ and RPT

By Lemma 1, a proof in RPT₀ 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 RPT₀ are weakly normalizing.

Theorem 7 RPT₀ is consistent. The subformula property and the disjunction property hold for RPT₀.

As we stated before, RPT₀ is not exactly equivalent to RPT without inductive definition. But the structure of proof-figures of RPT is the same as that of RPT₀, 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 RPT₀.

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 RPT. By putting three reasonable conditions, RPT does correspond to RPT. We have studied other constructive logics: Martin-Löf’s type theory ITT₀[29], Feferman’s T₀[18], Hayashi and Nakano’s P,X[21], Sato’s SST[35], and Coquand and Huet’s Calculus of Constructions (CoC)[11]. ITT₀ and CoC have type theories while T₀, P,X, and 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, we 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 RPT₀ and RPT₅. Every proof-figure of RPT₀ is transformed into a proof-figure of RPT₅ and an application of the equal-left rule. The calculus of RPT₅ 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 Red₀(p) satisfies it. From this theorem, we got several fundamental proof-theoretic properties for RPT₅, including the strong normalization property in a restricted calculus, the consistency, and the subformula property. We proved the similar properties for RPT₀, 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 \( \Lambda \). We also chose \( \Lambda \) as the implementation language of our system, so the two languages are identical in our case.
CHAPTER 3. CONSTRUCTIVE PROGRAMMING SYSTEM

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 proof 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 identity. 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 (\_  \_ ) 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 |- 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).

<table>
<thead>
<tr>
<th>RPT</th>
<th>Input to the system</th>
<th>Output from the system</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>false</td>
<td>⊥</td>
</tr>
<tr>
<td>t</td>
<td>true</td>
<td>t</td>
</tr>
<tr>
<td>a = b</td>
<td>(eqv a b)</td>
<td>a = b</td>
</tr>
<tr>
<td>p(a)</td>
<td>(pred p a)</td>
<td>p(a)</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>(and A B)</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>(and A B)</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>A \lor B</td>
<td>(or A B)</td>
<td>A \lor B</td>
</tr>
<tr>
<td>A \supset B</td>
<td>(imp A B)</td>
<td>A \supset B</td>
</tr>
<tr>
<td>\forall x. A</td>
<td>(all (lambda (x) A))</td>
<td>\forall x. A</td>
</tr>
<tr>
<td>\exists x. A</td>
<td>(ex (lambda (x) A))</td>
<td>\exists x. A</td>
</tr>
<tr>
<td>p</td>
<td>- p</td>
<td>= p = p</td>
</tr>
</tbody>
</table>

Table 3.1: Logical Connectives in Input and Output

3.2. OVERVIEW OF THE SYSTEM

Our system is an interactive proof generator, not a proof-checker (which checks a proof after inputting 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 the 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 |- T true. The expression Result:100: is the header of this reply and the actual content is 0 |- T only, which is the righthand side of the sequent 0 |- T 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 \(\lambda\)-introduction rule to the subproofs numbered 70 (two steps before this application) and 71 (one step before this application).

\[
\text{RPT:72} \quad \text{and-intro} \quad -2 \quad -1
\]

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

\[
\text{RPT:72} \quad \text{and-intro}
\]

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 \text{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.

\[
\text{RPT:74} \quad \text{prop-eql} \quad \text{term?}
\]

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 \(\lambda\)-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 \(\forall\). 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. OVERVIEW OF THE SYSTEM

3.2.2 Automatic Proof Generation

Although our system does not aim to generate proofs automatically, 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, \(a \supset b\) 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 \(\lambda\) 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, \(\text{true} \land A\) can be simplified to \(A\), and \(\text{car}(A, B)\) to \(A\).

We then describe \(E\)-simplification.

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

\[
\begin{align*}
\varepsilon_1 & \triangleq \exists x. \exists y. \exists z. (u = \langle x, y \rangle) \land y = \langle 0, z \rangle \land P[z])
\end{align*}
\]

Let \(p\) be a proposition, and \(e\) be an \(E\)-proposition. Suppose \(a \vdash p\) and \(b \vdash e\) have already been proved. If the following procedure succeeds with a new proposition \(r\), then we obtain a proof of \(a \vdash r\) 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 \(\theta\) 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
\[
pair?(u) = \text{true} \land \text{fun?}(cdr(u)) = \text{true} \lor p_2.
\]
We assume that \( p_2 \) does not contain \( u \) free. The most general unifier of the set of equalities in \( e_1 \) is
\[
\theta_i \triangleq \{ u := (x,(0,z)), y := (0,z) \}
\]
Then \( p_1 \theta \) is
\[
pair?(\langle x,(0,z) \rangle) = \text{true} \land \text{fun?}(cdr(\langle x,(0,z) \rangle)) = \text{true} \lor p_2,
\]
and 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-propsitions 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 \( \Lambda \).

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 \( bld \) and \( cld \) 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 \( bld \) and \( cld \) 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 \( \Lambda \) 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 \( \Lambda \) 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 \( \Lambda \). The method is summarized 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 \((\lambda x.xx)(\text{car}(y,z)))\), then the result is one of this term itself, \((\text{car}(y,z))(\text{car}(y,z))\), \((\lambda x.xx)\), 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,
\[
((\lambda x.xx)(\text{car}(y,z)))^* = yy
\]

3. Prove the following properties on \( \rightarrow \) and \( \rightarrow^* \).
   * \( a \rightarrow a \)
   * \( a \rightarrow c \) and \( b \rightarrow d \) imply \( a[c]b \rightarrow c[d] \)
   * \( a \rightarrow b \) 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 \( \Lambda \) as pairs.

All the terms in \( \Lambda \) are already represented as pairs by the **quote** mechanism. But we must again represent the terms of \( \Lambda \) 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.
CHAPTER 3. CONSTRUCTIVE PROGRAMMING SYSTEM

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 with 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 \( \Lambda \) 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 \( a \):

\[
\forall x. (\text{Unary}(S) \supset \text{Binary}(A) \supset \text{Persis}(A, S) \supset \\
\text{Diamond}(A, S) \supset \text{Diamond}(	ext{Trans}(A), S))
\]

where

- \( \text{Unary}(S) \equiv \forall x. \exists y \ldots S[x] \)
- \( \text{Binary}(A) \equiv \forall x. \forall y. \exists z \ldots A[x, y] \)
- \( \text{Trans}(A) \equiv \lambda f, \lambda x. \lambda y. \ldots f[x, y] \)
- \( \text{Persis}(A, S) \equiv \lambda x, y. \forall z \ldots A[x, y] \supset S[z] \supset S[y] \)
- \( \text{Diamond}(A, S) \equiv \forall x, y. \forall z \ldots A[x, y] \supset A[x, z] \supset S[z] \supset S[y] \)

\[ \square \]

In this theorem, the unary predicate \( S \) represents the (quote'd) termhood, and \( \text{Persis}(A, S) \) 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 \( \text{Trans}(A) \) 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 this 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:

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

Here two predicates **List** and **Append** are defined as follows:

- \( \text{List} \triangleq \lambda f, \lambda x. (x = \text{nil} \lor \exists x_1, x_2. x = (x_1, x_2) \land f[x_2]) \)
- \( \text{Append} \triangleq \lambda f, \lambda x, y, z. x = \text{nil} \land y = z \lor \exists x_1, x_2. (x = (x_1, x_2) \land \exists z_1. (f[x_2, y, z_1] \land z = (x_1, z_1))) \)

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:

\[
\mu(\lambda x. q). \\
(\lambda r. \text{if car}(r) \text{ then } \lambda y. [y \text{ true 0 } 0] \text{ else } y]) \\
[\text{false \ if caddr(r) . cadr(cdddr(r))yl} \text{ else } \text{caddr(r)}] \\
\text{cdr(cdddr(r))yl} \text{ . 0}] \\
\text{fi}) \\
[\text{car(q)} . \\
\text{if car(q) then cdr(q) \ else } \text{cadr(q)} . \text{cdddr(q)} . \text{f(cdddr(q))(cdddr(q))}] \\
\text{fi}]
\]

This program is the proof-term of the specification above.
The term \( T_{app} \) requires four arguments:

- A proof \( x \).
- A proof \( y \) that \( x \) is a list.
- A proof \( y \).
- A proof \( l \) that \( y \) is a list.

The return value of \( T_{app} \) is a proof of \( \exists: \text{Append}(x, y, z) \). Therefore, the result of appending two lists \( x \) and \( y \) is \( \text{car}(T_{app}(r_{list}(x), y(r_{list}(y)))) \) 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 \( a \):

\[
\alpha \vdash \{ \text{List}(x) \supset r_{list}(x) \vdash_0 \text{List}(x) \}
\]

where \( r_{list} \) is defined as follows:

\[
r_{list} \triangleq \mu(\lambda x. \\
\quad \text{if } \text{null}(x) \text{ then } (\text{true},0) \\
\quad \text{else } \text{false}, \text{car}(x), \text{cdr}(x), (0, f(\text{cdr}(x)))) \}
\]

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 \( \Lambda \) is defined similarly as this extensional equality. However, our definition is more complex than this, since the computation of \( \Lambda \) is call-by-name rather than call-by-value, and the values are canonical terms rather than normal terms. For instance, let \( \eta \) be the following term:

\[
(\mu(\lambda f. \lambda n. \{n, f(suc(n))\}))0
\]

where \( \text{suc}(n) \) represents the successor of \( n \). The term \( I \) represents an infinite list of natural numbers. Both terms \( \Omega \) and \( I \) 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 \( \text{pair}(x) \), 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 \( \Lambda \) 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:

\[
s T[R] t \triangleq \\
( s \rightarrow^* \text{nil} \supset t \rightarrow^* \text{nil} ) \\
\quad \land ( s \rightarrow^* \text{true} \supset t \rightarrow^* \text{true} ) \\
\quad \land ( s \rightarrow^* \text{false} \supset t \rightarrow^* \text{false} ) \\
\quad \land ( s \rightarrow^* (s_1, s_2) \supset (t \rightarrow^* (t_1, t_2) \land (s_1 R t_1) \land (s_2 R t_2)) ) \\
\quad \land ( s \rightarrow^* \lambda x. a \supset \exists b. (t \rightarrow^* \lambda y. b \land u_0 \rightarrow (a_0, b_0[u_0])) )
\]

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 \( R \) in \( T[R] \).

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 \( \{x_1, \ldots, x_n\} \). Then \( F \leq G \) if, for any closed terms \( d_1, \ldots, d_n, F[d_0] \leq G[d_0] \).

As an example of \( \leq \), for any term \( t \), we have \( \Omega \leq t \). We also have \( I \leq (0, (x, y)) \).

**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.\( \Box \)

**Theorem 10** We have the following:

1. \( \leq \) is reflexive and transitive.
2. if \( A \equiv B \) implies \( a \leq b \).
3. \( \sim \) is an equivalence relation. \( \Box \)

\(^1\)In this definition, the logical connectives \( \land \) and \( \lor \) are not formal ones in RPT.
Theorem 11 Let \( x, y, z, u, v, w \) be variables, and \( a, b \) be terms. Let \( \text{fun} \) be one of \( \text{car}, \text{cdr}, \text{null}?, \text{true}? \), \( \text{false}? \), \( \text{pair}? \), or \( \text{fun}? \). Then we have
\[
\text{fun}(\text{if } x \text{ then } y \text{ else } z \text{ if } i) \sim \text{if } x \text{ then } \text{fun}(y) \text{ else } \text{fun}(z) \text{ if } i,
\]
\[
(\text{if } x \text{ then } y \text{ else } z \text{ if } i)(w) \sim \text{if } x \text{ then } y(w) \text{ else } z(w) \text{ if } i,
\]
\[
(\text{if } x \text{ then } y \text{ else } z \text{ if } i) \text{ then } u \text{ else } v \text{ if } i \sim \text{if } x \text{ then } (\text{if } y \text{ then } u \text{ else } v) \text{ if } i,
\]
and
\[
\text{if } x \text{ then } a \text{ else } b \text{ if } i \sim \text{if } x \text{ then } \text{a, if true? else \text{b, if false? else \text{if } i}}.
\]

Proof.
We will prove the case for \( \text{car} \) of the first equation only.
Let \( \theta \) be a substitution which substitutes closed terms for \( x, y, z \).
\[
L \triangleq \text{car}(\text{if } x \text{ then } y \text{ else } z \text{ if } i)\theta
\]
\[
M \triangleq (\text{if } x \text{ then } \text{car}(y) \text{ else } \text{car}(z) \text{ if } i)\theta
\]
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 \( \Lambda \), we have either (i) \( x\theta \to^* \text{true?} \) and for some terms \( a, b, y\theta \to^* (a, b) \), or (ii) \( x\theta \to^* \text{false?} \) and for some terms \( a, b, x\theta \to^* (a, b) \). 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 \( \equiv \). Hence, \( F \sim G \) does not imply \( F \equiv G \).

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:
\[
c ::= f
\]

Two terms \( a \) and \( b \) are observationally equivalent if, for any context \( C() \), \( C(a) \) and \( C(b) \) reduce to canonical forms of the same kind, or they do not reduce to canonical forms.

3.4. PROGRAM SYNTHESIS

\[
\begin{array}{l}
\text{null?}(c) | \text{true?}(c) | \text{false?}(c) \\
\text{pair?}(c) | \text{fun?}(c) \\
\text{car}(c) | \text{cdr}(c) \\
c(i) \\
\text{if } c \text{ then } t \text{ else } t \text{ fi}
\end{array}
\]

where \( t \) is a term.

Theorem 12 Suppose the following hold for a term \( F \), mutually distinct variables \( f, x_1, \ldots, x_n \), terms \( c, a, b_i \) \((1 \leq i \leq n, 1 \leq j \leq k)\):
\[
FV(F) = \{f\} \\
FV(a) = \emptyset \\
FV(b_i) \subseteq \{x_1, \ldots, x_n\} \\
FV(c) \subseteq \{f, x_1, \ldots, x_n\}
\]
\( c \) is strict with respect to \( f \)
\( c_f[F] \sim a(c_f[\theta_1], \ldots, c_f[\theta_k], \bar{x}) \)

Then the following holds.
\[
c_f[\mu(\lambda f. F)] \sim G(\bar{x})
\]
where \( G \) is defined as follows:
\[
G \triangleq \mu(\lambda g. \lambda \bar{x}. a(g(\bar{\theta}_1), \ldots, g(\bar{\theta}_k), \bar{x}))
\]

Theorem 12 plays a central role when we eliminate redundancy in recursive functions which uses \( \mu \). For instance, let \( F \) be the following term:
\[
\lambda x. \text{if } \text{null?}(x) \text{ then } \{b, c\} \\
\text{else } [p(\text{car}(f(\text{pred}(x))))], q(f(\text{pred}(x)))] \text{ fi}
\]
where \( \text{pred}(x) \) 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 \( F'(\text{pred}(n)) \). Therefore, if we want to have the first component of the pair, namely, \( \text{car}(F'(n)) \), then we do not have to compute the second component. Formally speaking, let \( H \) be the following:
\[
\lambda x. \text{if } \text{null?}(x) \text{ then } b \\
\text{else } p(g(\text{pred}(x))) \text{ fi}
\]
CHAPTER 3. CONSTRUCTIVE PROGRAMMING SYSTEM

Intuitively, we can replace \( \text{car}(\mu F)x \) by \( \mu(G.H)x \). We do not have
\[
\text{car}(\mu F)x \approx \mu(G.H)x
\]
however, we have
\[
\text{car}(\mu F)x \sim \mu(G.H)x
\]
hence this replacement is justified with respect to \( \sim \).

This example is an instance of Theorem 12 where \( c, F \) and \( G \) are \( \text{car}(f x).F \) and \( \mu(G.H) \). Also we used Theorem 11 in the transformation of
\[
c/F \sim \text{if } \text{null?}(x) \text{ then } b \text{ else } \mu(\text{car}(f (\text{pred}(x))))\text{ fi}
\]

\[\text{Theorem 13}\]
For mutually distinct variables \( f, y, x_1, \ldots, x_m \), terms \( F, G \), \( a, b, c \) \((1 \leq i \leq m, 1 \leq j \leq k)\), if the following holds:
\[
\begin{align*}
F & \equiv \lambda x.\lambda y.a(f(\bar{b}^i), \ldots, f(\bar{b}^j), \bar{x}) \\
G & \equiv \mu(F) \\
FV(a) & = \emptyset \\
FV(b^i) & \subseteq \{x_1, \ldots, x_m\} \\
FV(c^j) & \subseteq \{x_1, \ldots, x_m, y\}
\end{align*}
\]

then we have
\[
G(\bar{x}, y) \sim \mu(\lambda x.\lambda y.a(g(\bar{b}^i), \ldots, g(\bar{b}^j), \bar{x}))(\bar{x})
\]
\[\square\]

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 \( \mu \) in \( T_{\text{app}} \), and \( c \) be \( \text{car}(f x(r_{\text{ref}}(x)y)l) \). Then, by using Theorem 11, the term \( c/F \) can be simplified, and then we have the following by Theorem 12:
\[
\begin{align*}
\text{car}(T_{\text{app}}(r_{\text{ref}}(x)y)l) & \sim \mu(\lambda x.\lambda y.l, \text{if } \text{null?}(x) \text{ then } y \text{ else } (\text{car}(x), g(\text{cdr}(x), y, l)) \text{ fi}) \\
\end{align*}
\]
In the right-hand side of the above equation, the variable \( l \) is not used. Hence by Theorem 13, we have:
\[
\begin{align*}
\text{car}(T_{\text{app}}(r_{\text{ref}}(x)y)l) & \sim \mu(\lambda x.\lambda y. \text{if } \text{null?}(x) \text{ then } y \text{ else } (\text{car}(x), g(\text{cdr}(x), y)) \text{ fi}) \\
\end{align*}
\]
The last definition is the same as the usual hand-written program of append.

### 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 Fereman's T0, 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 \( \Delta \) 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: |- \( \forall x. (1 = S(x)) \)

; Assume this proposition
Result: 3: \( \forall x. \forall y. (A[x,y]) \)

; Assume this proposition

Result: 5: \( \forall x. \forall y. (A[x,y] \supset S[x] \supset S[y]) \)

; Assume this proposition

Result: 14: \( \forall x. (S[x] \supset \forall z. \forall y. (A[x,y] \land A[x,z] \supset \exists u. (A[y,u] \land A[z,u]))) \)

; Assume this proposition

\[ \text{RPT: 21} \]
(defindpred TransA (x y) it X)

pred-name: TransA
pred-body: \( x = y \lor \exists z. (A[x,z] \land TransA[z,y]) \)

OK

; Define the predicate TransA

\[ \text{RPT: 26} \]
(name BasicProp1 (all-intro it w))

Result: 26: \( \forall w. TransA[w,w] \)

; Name the proof of this proposition as BasicProp1

\[ \text{RPT: 35} \]
(name BasicProp2 (all-intro (all-intro it y) x))

Result: 35: \( \forall x. \forall y. (A[x,y] \supset TransA[x,y]) \)

; Prove \( x \rightarrow y \supset x \rightarrow y \)

\[ \text{RPT: 69} \]
(defpred Prop1 (x y) it)

pred-name: Prop1
pred-body: \( S[x] \supset \forall z. (A[x,z] \supset \exists u. (TransA[y,u] \land TransA[z,u])) \)

OK

; Name the proof of this proposition as Prop1

\[ \text{RPT: 80} \]
(name IH (assume (prop-and (prop-pred A (x z)) (prop-pred Prop1 (y z))) a3))

Result: 80: \( A[x,z] \land Prop1[z,y] \)

; Induction Hypothesis in the proof of Prop1

\[ \text{RPT: 117} \]
(name Proof-of-Prop1)

Result: 117: \( \forall x. \forall y. (TransA[x,y] \supset Prop1[x,y]) \)

\[ \text{RPT: 123} \]
(defpred Prop2 (x y) it)

pred-name: Prop2
pred-body: \( S[x] \supset \forall z. (TransA[x,z] \supset \exists u. (TransA[y,u] \land TransA[z,u])) \)

OK

3.5. CONCLUSION

; Define the above predicate Prop2

\[ \text{RPT: 133} \]
(name IH2 (assume (prop-and (prop-pred A (x z)) (prop-pred Prop2 (y z))) a3))

Result: 133: \( A[x,z] \land Prop2[z,y] \)

; Induction Hypothesis in the proof of Prop2

\[ \text{RPT: 175} \]
(name Proof-of-Prop2)

Result: 175: \( \forall x. \forall y. (TransA[x,y] \supset Prop2[x,y]) \)

; 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 contain 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 \textit{positive} inductive definitions as shown below. We call a pair of the following two relations a \textit{provability relation}.

\[
\begin{align*}
A \text{ is a proposition (written as } \text{Prop}(A)) \quad &\equiv \quad a \text{ is a proof of } A \text{ (written as } \text{Proves}(a,A)) \\
\end{align*}
\]

In Martin-Löf’s type theory[29], these relations correspond to the judgements \( \text{A Set} \) 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 \( \text{Prop}(x) \) and \( \text{Proves}(x,y) \) naturally, they look like:

\[
\begin{align*}
\text{Prop}(y) &\leftrightarrow \ldots \quad \exists u. \exists v. \text{Prop}(u) \land y = (u \supset v) \land \forall e. (\text{Proves}(e,u) \supset \text{Prop}(v)) \\
\text{Proves}(x,y) &\leftrightarrow \ldots \quad \exists u. \exists e. \text{Prop}(u) \land y = (u \supset v) \\
&\quad \land \forall e. (\text{Proves}(e,u) \supset \text{Proves}(e,v))
\end{align*}
\]

where \( \supset \) 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 \textit{monotone}. However, in the above formulation, an occurrence of \( \text{Proves}(e,u) \) 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 \textit{positive} inductive definitions.

Several researchers have attacked this problem in different contexts. Bescs[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 \( \text{Proves} \) and its negation \( \text{Proves} \) by a monotone inductive definition, and then \( \neg \text{Proves}(x,y) \leftrightarrow \text{Proves} \) 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 \textit{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.

Dybjær[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 \textit{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
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)

\[ t ::= x \mid c \mid c?() \mid \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \text{ fi} \]

where \( x \) is a variable and \( c \) is a constant.

Terms \( (t_1, t_2) \), \( \text{car}(t) \), and \( \text{cdr}(t) \) correspond to \( \text{cons} \), \( \text{car} \), and \( \text{cdr} \) in Lisp. We use the standard abbreviation for lists; \( (t) \) for \( (t_1, \text{nil}) \), and \( (t_2) \) for \( (t_1, (t_2, \text{nil})) \) and so on. Terms \( \lambda x.t \) and \( \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \text{ fi} \) are as usual. Terms of the form \( c \), \( (t_1, t_2) \), or \( \lambda x.t \) are called canonical. For a canonical term \( t \), the terms \( c?(t) \), \( \text{pair}?(t) \) and \( \text{fun}?(t) \) are equal to true if they are in either form of \( c?() \), \( \text{pair}?(s, u) \) or \( \text{fun}?() \), and equal to false otherwise. We assume a call-by-name evaluation mechanism; for example, \( \text{car}((\text{nil}, t)) \) is equal to \( \text{nil} \) regardless of \( t \). A term of the form \( q_i[t] \) 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)

\[ F ::= \perp \mid t_1 = t_2 \mid t \downarrow \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid F_1 \supset F_2 \mid \forall x.F \mid \exists x.F \mid A^n(t_1, \ldots, t_n) \]

\[ A^n ::= F \]

\[ A^n ::= X^n \]

\[ \lambda x.A^n \]

\[ \mu X^n.A^n \]

(\text{for each } n > 0)

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 \( \perp, t_1 = t_2, \) or \( t \downarrow \) is called atomic. A \( \perp \) means that the term \( t \) has a canonical form. The formula \( A^n(t_1, \ldots, t_n) \) represents the application of \( n \) terms \( t_1, \ldots, t_n \) 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 \( x_1, \ldots, x_n \) as \( x \) and a list of \( n \) terms \( t_1, \ldots, t_n \) as \( t \). Similarly, the formula \( A(t_1, \ldots, t_n) \) is abbreviated as \( A(t) \). \( \neg A \) and \( A \leftrightarrow B \) are abbreviations for \( A \supset \perp \land (A \supset B) \land (B \supset A) \), 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 \( \land, \lor, \supset, \forall, \exists \) (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
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 $\text{Prop}$ and $\text{Proves}$. 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 $(\text{Prop}, \text{Proves})$ has some kind of monotonicity; $\text{Prop}(A \land B)$ is defined by using $\text{Prop}(A)$ and $\text{Prop}(B)$, which are already defined. $\text{Prop}(A \lor B)$ is defined by using $\text{Prop}(A)$, $\text{Prop}(B)$, and $\text{Proves}(x, A)$. Since the set of the $\forall$ satisfying $\text{Proves}(r, A)$ is defined at the time of defining $\text{Prop}(A)$ and will never be changed after defined, we may consider that $\text{Prop}(A \lor B)$ 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 $(P, Q)$ as follows:

\[
(P_0, Q_0) \leq_R (P_1, Q_1) \iff \forall x. (P_0(x) \supset P_1(x)) \land \forall x, y. (Q_0(x, y) \supset Q_1(x, y))
\]

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 $(\text{Prop}, \text{Proves})$ 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-positivity[5].

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

\[
(S_0, T_0) \leq_A (S_1, T_1) \iff S_0 \subseteq S_1 \land \forall x \in S_0. (x \in T_0 \iff x \in T_1)
\]

It is easily seen that $\leq_A$ is a special case of $\leq_R$. For a pair of sets $(S_i, T_i)$, we define $P_i(x) \triangleq x \in S_i$ and $Q_i(x, y) \triangleq x \in T_i \land y = 0$. Then $(S_0, T_0) \leq_A (S_1, T_1)$ are equivalent to $(P_0, Q_0) \leq_R (P_1, Q_1)$ provided that $T_i \subseteq S_i$ 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 $x$ be a list of $n$ distinct variables. Then we define

\[
P_0 \leq_N P_1 \iff \forall x. P_0(x) \supset P_1(x)
\]

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

\[\text{MONO}(A; X) \triangleq X_0 \leq_N X_1 \supset A_{X[X_0]} \leq_N A_{X[X_1]}\]

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

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

The mechanism of monotone inductive definitions (MID, in short) is that, for any $A$ satisfying $\text{MONO}(A; X)$, we have the least solution of $\forall x. (A(x) \leftrightarrow X(x))$. 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 $\text{MONO-Q}$, and allowed MID only when $\text{MONO-Q}(A; X)$ is satisfied. Roughly speaking, $\text{MONO-Q}(A; X)$ is the condition that the relation "$\exists$ realizes $A(x)$" is monotone in the relation "$\exists$ realizes $X(x)$". Since $\text{MONO}(A; X)$ does not subsume $\text{MONO-Q}(A; X)$, this is a proper restriction.

The rules for MID, a restricted version of MID, are as follows.

\[
\text{MONO}(A) \quad \text{MONO-Q}(A)
\]

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

**Remarks**

\( \text{MID}_0 \) can be extended to a simultaneous inductive definition straightforwardly. 

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

\[
\forall i \leq n \exists j \leq n (P_i = Q_j) \quad \text{and} \quad \forall X \subseteq \{1, \ldots, n\} | P_i = Q_j \Rightarrow \forall x \in X (P_i(x) \Rightarrow Q_j(x))
\]

where \( R^n \) and \( Q^n \) are \( \lambda X. \bot \).

If \( df(i) = 0 \), the condition becomes \( P^i \leq Q^i \). Note that, the orders \( \leq N \) and \( \leq R \) can be written as \( \leq R_{1,2,3}(i) \) and \( \leq R_{1,2,3}(i) \) where \( df(1) = 0, df(2) = 0, df(3) = 1 \) and \( df(4) = 3 \). Then the order \( \langle P_0, Q_0, R_0, S_0 \rangle \leq R \langle P_1, Q_1, R_1, S_1 \rangle \) 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 \( R \)-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, \( \mu X_i(A_1, \ldots, A_n; X_1, \ldots, X_n) \) is an \( n \)-ary abstract with \( X_1, \ldots, X_n \) bound by this \( \mu \) 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 \( (A, B) \) 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:

\[
\text{HMONO}(A, B; X, Y) \triangleq \text{HMONO}(A, B; X, Y) \triangleq \{X_0, Y_0 \} \leq_R (X_1, Y_1) \supset \{A_{X, Y}[X_0, Y_0], B_{X, Y}[X_0, Y_0] \} \leq_R \{A_{X, Y}[X_1, Y_1], B_{X, Y}[X_1, Y_1] \}
\]

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

Let us abbreviate as follows:

\[
M_X \triangleq \mu X. (A, B; X, Y) \\
M_Y \triangleq \mu Y. (A, B; X, Y) \\
\mu - \text{eq} \triangleq \forall X. (M_X(x) \leftrightarrow A[M_X, M_Y](x)) \land \forall y. (M_Y(x, y) \leftrightarrow B[M_X, M_Y](x, y))
\]

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

\[
\frac{\mu - \text{eq}}{\text{HMONO}(A, B)} (\mu - \text{eq} - 0)
\]

\[
\frac{\text{HMONO}(A, B) \quad \{A[F, G], B[F, G] \} \leq_R (F, G)}{(M_X, M_Y) \leq_R (F, G)} (\mu - \text{ind} - 0)
\]

where \( F \) and \( G \) are arbitrary \( n \)-ary and \( n+m \)-ary predicates.

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

Similarly, we can define the \( \mu - \text{eq} \) and \( \mu - \text{ind} - 0 \) rules with respect to an arbitrary order \( \leq_{R(x, y)} \).

The union of \( \mu - \text{eq} \) and \( \mu - \text{ind} - 0 \) for an arbitrary order \( \leq_{R(x, y)} \) is called HMID.

Since the order \( \leq_{R(x, y)} \) is an extension of \( \leq_X \), 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 \( \text{HMONO-Q} \) is a corresponding condition to \( \text{MONO-Q} \) in \( \text{MID}_0 \). The second one is that, we can use the \( \mu - \text{ind} \) 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:

\[
\frac{\mu - \text{eq}}{\text{HMONO}(A, B)} (\mu - \text{eq} - 1)
\]

\[
\frac{\text{HMONO}(A, B) \quad \{A[F, M_Y], B[F, M_Y] \} \leq_R (F, M_Y)}{(M_X, M_Y) \leq_R (F, M_Y)} (\mu - \text{ind} - 1)
\]

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

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

Under the conditions \( \text{HMONO} \) and \( \text{HMONO-Q} \), the first rule states that \( (M_X, M_Y) \) is a solution, and the second one states that it is the least one. Since \( \mu - \text{ind} - 1 \) 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 \( M_Y(x, y) \) where \( M_X(x) \) does not hold, since such an \( x \) is outside of the domain. In Sections 4.5 and 4.6, we shall show the above formulation is sufficient for most purposes.

The \( \mu - \text{ind} - 1 \) rule subsumes the following more convenient form:

\[
\frac{\text{HMONO}(A, B) \quad \text{HMONO-Q}(A, B) \quad A[F, M_Y] \leq_R F \quad F \leq_R M_X}{M_X \leq_R F} (\mu - \text{ind} - 0)
\]

The union of the \( \mu - \text{eq} \) and \( \mu - \text{ind} - 1 \) rules (for \( \leq_R \) order) is called \( \text{HMID}_0 \). Note that \( \text{HMID}_0 \) is an extension of MID.

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 \( \leq_R \): 

\[
\forall x, y. \quad P_0(x) \supset Q_1(x, y) \supset Q_0(x, y)
\]

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 \( \text{BT+HMID} \).

We begin with an arbitrary model of \( \text{BT} \); 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 \( m_1, \ldots, m_n \), we define an order \( \Xi_{n, df} \) on \( Dom \) as follows, where \( Dom \) is \( \mathcal{P}(D^{m_1}) \times \cdots \times \mathcal{P}(D^{m_n}) \).
Definition 25 (Order \(\sqsubseteq_{s,df}\)) For \(S_i \subseteq D^{m_i}\) and \(T_i \subseteq D^{m_i}\) (\(1 \leq i \leq n\)), we define \((S_1, \ldots, S_n) \sqsubseteq_{s,df} (T_1, \ldots, T_n)\) if for each \(i\) with \(1 \leq i \leq n\), the conjunction of the following conditions hold:

1. \(S_{i|g(0)} \subseteq T_{i|g(0)}\)
2. \(S_i \subseteq T_i\)
3. \(T_i[S_{i|g(0)} = S_i|S_{i|g(0)}]\)

where \(T[S] \triangleq \{(s, t) \mid s \in S \land (s, t) \in T\}\)

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

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

\[\langle S_1, \ldots, S_n \rangle \sqsubseteq_{s,df} \langle T_1, \ldots, T_n \rangle \text{ implies } F(S_1, \ldots, S_n) \subseteq_{s,df} F(T_1, \ldots, T_n)\]

Lemma 3 The following holds:

1. \(\sqsubseteq\) is an order on \(\text{Dom}\).
2. The least element exists in \(\text{Dom}\). Moreover, every directed subset of \(\text{Dom}\) 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 \(\text{Dom}\) with \(\sqsubseteq\) is a complete partial order (cpo), which implies the third clause (in classical set theory).

As for the second clause: let \(\Delta\) be a directed set in \(\text{Dom}\). Let \(L_i\) be 

\[\cup\{X_i \mid \langle X_1, \ldots, X_n \rangle \in \Delta\}\].

We will prove that \((L_1, \ldots, L_n)\) is the lub of \(\Delta\).

Suppose it is not an upper bound of \(\Delta\). Then there exists \((S_1, \ldots, S_n) \in \Delta\) such that \((S_1, \ldots, S_n) \not\subseteq (L_1, \ldots, L_n)\) does not hold. From the definition of \(L_i\), we have that \(L_i[S_{i|g(0)}]\) is properly larger than \(S_i[S_{i|g(0)}]\) for some \(i\). Then we have that there exists \((T_1, \ldots, T_n) \in \Delta\) such that \(T_i[S] = T_i[S]\) is properly larger than \(S_i[S]\) for some \(i\) and some \(S \in S_{i|g(0)}\). But this contradicts the directedness of \(\Delta\), since we do not have an upper bound of \((S_1, \ldots, S_n)\) and \((T_1, \ldots, T_n)\). The minimality is apparent from the definition.

We can also prove that \((\emptyset, \ldots, \emptyset)\) is the least element in \(\text{Dom}\).

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 \((\mathbb{D}^{m_1}, \ldots, \mathbb{D}^{m_n})\) is not always greater than an arbitrary element in \(\text{Dom}\).

We can interpret a monotone operator in \(BT + \text{HMID}\) 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 \(\mu\)-eq-0 and \(\mu\)-ind-0 are satisfied. As a consequence, we have the following theorem.

Theorem 14 (Consistency)

1. We can construct a model of \(BT + \text{HMID}\) from any model of \(BT\).
2. \(BT + \text{HMID}\) (therefore, \(BT + \text{HMID}_0\)) 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

Let \(\text{Nat}\), the predicate for natural numbers, is defined as follows. Let \(A\) be the following template:

\[\lambda x. x = 0 \lor \exists y. (y = \text{succ}(x) \land X(y))\]

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

Prime Numbers

The next example is a predicate for prime numbers.

At first thought, it is defined by:

\[\text{Prime}(n) \iff \text{Nat}(n) \land 1 < n \land \forall m. (1 < m < n \Rightarrow \text{Div}(m,n))\]

where \(\text{Div}(m,n)\) means “\(m\) divides \(n\)”.

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

\[\text{Prime}(n) \iff \text{Nat}(n) \land 1 < n \land \forall m < n. (\text{Prime}(m) \Rightarrow \text{Div}(m,n))\]

Since the latter contains a negative occurrence of \(\text{Prime}\), it is not directly acceptable by conventional theories.

On the contrary, we will show that we can define it by \(\text{HMID}\) as follows.
4.4 Realizability interpretation and its soundness for BT+HMID₀

In this section, we define a realizability interpretation for BT+HMID₀ 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 \( \lambda x. e \ q X(x) \).

- Given a term \( e \) and a formula \( A \) in BT+HMID₀, the \( q \)-realizability \( e \ q A \) is a formula which is defined by induction on the structure of \( A \) as follows:

  1. \( e \ q A \triangleq e = \text{nil} \land A \) where \( A \) is atomic.
  2. \( e \ q Z(t) \triangleq Z(t) \land Z^*(t, e) \) where \( Z \) is a predicate variable.
  3. \( e \ q A \land B \triangleq (\text{car}(e) \ q A) \land (\text{cdr}(e) \ q B) \)
  4. \( e \ q A \lor B \triangleq (\text{car}(e)) \downarrow (\text{car}(e) = \text{true} \lor (\text{cdr}(e) \ q A)) \land (\text{car}(e) \neq \text{true} \lor (\text{cdr}(e) \ q B)) \)
  5. \( e \ q A \supset B \triangleq \forall x. A \land (r \ q A) \supset (e(r) \ q B) \)
  6. \( e \ q \forall x A \triangleq \forall x.(e(x) \ q A) \)
  7. \( e \ q \exists x A \triangleq (\text{cdr}(e) \ q A_0[\text{car}(e)]) \land A_0[\text{car}(e)] \)
  8. \( e \ q (\lambda x A)(t) \triangleq e \ q A_0[t] \)
  9. \( e \ q (\mu x. (\lambda x A). (\lambda x y B))(t) \triangleq (\mu x^*. (\lambda x y. \lambda x y A)(r \ q A)) \land x y B, \lambda x y s.(s \ q B); X, X^*, Y, Y^*) \downarrow (t, e) \)
  10. \( e \ q (\rho x. (\lambda x A). (\lambda x y B))(t, u) \triangleq (\mu x^*. (\lambda x A). (\lambda x y A)(r \ q A)) \land x y B, \lambda x y s.(s \ q B); X, X^*, Y, Y^*) \downarrow (t, u, e) \)

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 realization of \( (\mu x A)(t) \) and that of \( A_X[\mu x A](t) \) are the same. So we want to define \( e \ q (\mu x A)(t) \) as \( e \ q A_X[\mu x A](t) \). However, by expanding \( e \ q A_X[\mu x A](t) \), we shall encounter \( e \ q \mu x A \ q A \), which we are defining now. By defining \( X^*(t, e) \) as \( e \ q X(t) \), we have another inductive definition \( X^*(t, e) \supset (e \ q A(t)) \), 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

In this interpretation, the argument \( n \) is the overlapping argument for \( A \) and \( B \). We assume here that \( \lambda n \) and \( \text{Div} \) 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 \( \text{Nat} \), 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 \( (M_X, M_Y) \) for the equation \( (A(x), B(x)) \leftrightarrow (X(x), Y(x)) \). We define \( \text{Prime} \) as \( M_Y \). Using the induction rule, \( M_X(x) \leftrightarrow \text{Nat}(x) \) is easily proved, and we get

\[
\text{Prime}(n) \leftrightarrow \text{Nat}(n) \land 1 < n \land \forall m < n. (\text{Prime}(m) \supset \neg \text{Div}(m, n))
\]
as a desired result.

**Verification of the HMONO(A, B) condition**

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

The verification of the HMONO-Q(A, B) condition is postponed until Section 4.4.3.
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 $(e \alpha A)$ 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 $HMONO-Q(\lambda x A, \lambda xy B)$ is defined as the quadruple of abstracts $\langle \lambda x A, \lambda xy (r A q A, \lambda xy B, \lambda xy s q B) \rangle$ in monotone sort, $R_4$-order.

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

**Lemma 4** Assuming the $HMONO$ and $HMONO-Q$ conditions hold, we have $\langle L_0, L_1, L_2, L_3 \rangle \leq \langle M_0, M_1, M_2, M_3 \rangle$. Applying these elements to $\langle \lambda x A, \lambda xy (r A q A, \lambda xy B, \lambda xy s q B) \rangle$, we have

$$\langle L_0, \lambda xy (r A q A), L_1, \lambda xy s q B \rangle \leq \langle M_0, M_1, M_2, M_3 \rangle$$

where $\theta$ is $\{ X \mapsto L_0, X' \mapsto M_1, Y \mapsto L_1, Y' \mapsto M_2 \}$. By the definition of $\leq R_4$ and $(L_0, L_1) \leq \langle M_0, M_2 \rangle$, we can replace the right-hand-side by $(L_0, L_1, L_2, L_3)$, and get

$$\langle L_0, \lambda xy (r A q A), L_1, L_2, L_3 \rangle \leq \langle M_0, M_1, M_2, M_3 \rangle$$

Taking this formula as an assumption of the $\mu$-ind-1 rule (besides the $HMONO$ and $HMONO-Q$ conditions), we obtain $\langle L_0, L_1, L_2, L_3 \rangle \leq \langle M_0, M_1, M_2, M_3 \rangle$ by $\mu$-ind-1. This formula implies $\langle M_0, M_2 \rangle \leq \langle L_0, L_1 \rangle$.

**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 $A_1, \ldots, A_n, F$ be formulas. If

$$A_1, \ldots, A_n, F \vdash_{BT+HMID_0} F$$

holds, then we have

$$A_1, \ldots, A_n, (x_1 A_1), \ldots, (x_n A_n) \vdash_{BT+HMID} F \quad q \ F$$

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 $B$:

- $HMONO(A, B)$
- $HMONO-Q(A, B)$

Then, the $\mu$-eq-1 rule is realized.

**Proof.** By the induction on the abstracts $A$ and $B$, we can prove that the same term is a realization of both sides of the $\mu$-eq-1 rule.

**Lemma 6** The $\mu$-ind-1 rule is realized.

**Proof.** Suppose that the following hold for $A$ and $B$.

- $HMONO(A, B)$
- $m \ q HMONO(A, B)$
- $HMONO-Q(A, B)$

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

$$\langle A[F, M_Y], B[F, M_Y] \rangle \leq \langle F, M_Y \rangle$$

By $\mu$-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

$$\forall x r. \langle A[F, M_Y] \wedge (r A q A[F, M_Y]) \supset (\text{car}(a) x r q F(x)) \rangle$$

(4.1)

Since we assume $HMONO-Q$, the quadruple $\langle \lambda x A, \lambda xy. r A, \lambda xy. s q B \rangle$ 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:

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

for an appropriate term $f$.

Let $H$ be $\lambda x, r. (L_X(x, r) \wedge (f x r q F(x)))$. Applying the quadruple $(M_X, H, M_Y, L_Y)$ to the $\mu$-ind-0 rule (in the case of $R_4$-order) with a little calculation, it becomes:
\[
\frac{\lambda x. (r \ q \ A[X,M_Y]) \theta \leq_N H}{L_X \leq_N H}
\]

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

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

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

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

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

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

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

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

\[
X_0 \triangleq M_X
\]

\[
X_1 \triangleq H
\]

\[
X_1^* \triangleq F
\]

\[
X_1^* \triangleq \lambda x. (M_X(x) \land L_X(x,r) \land (r \ q \ F(x)))
\]

\[
Y_i \triangleq M_Y (i = 0,1)
\]

\[
Y_i^* \triangleq L_Y \ (i = 0,1)
\]

we get \( M_X(x) \land L_X(x,r) \supset (t \ q A) \phi \) where

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

and

\[
(\lambda \ (c\text{ar}(m((f, \lambda x y s, s, \lambda x y r.s)))) x) r
\]

By the simple monotonicity in \( X_0 \), we may replace \( \phi \) by

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

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

By these lemmas, we have the soundness theorem. We also have the following corollaries.
This proof did not use the components of \( A \) and \( B \) in detail; it only mentioned the occurrences of \( X \) and \( Y \). That the \( \text{HMONO}(A, B) \) and \( \text{HMONO-Q}(A, B) \) 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 \( X(x) \land Y(x, y) \supset \cdots \) 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 \( \text{HMONO} \) and \( \text{HMONO-Q} \) conditions.

One may think the verification of the \( \text{HMONO-Q} \) 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 \( \text{HMD}_{\ell} \) mechanism.

#### 4.5.1 Defining a Provability Relation

We shall simultaneously define a unary predicate \( \text{Prop} \) and a binary predicate \( \text{Proves} \) which formalizes the following two concepts.

\[ \text{Prop}(p) \text{ represents } "p \text{ is a proposition}" \]
\[ \text{Proves}(a, p) \text{ represents } "a \text{ is a proof of } p" \]

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, \( a \land b \) is \( (c, a \land b) \) where \( c_a \) is a constant uniquely associated with \( \land \).

\[ A \triangleq \lambda x. \quad x = \perp \]
\[ \lor \exists a, b. \quad x = a = b \]
\[ \lor \exists a. \quad x = \hat{N}(a) \]
\[ \lor \exists y, z. \quad x = (y \land z) \land X(y) \land X(z) \]
\[ \lor \exists y, z. \quad x = (y \lor z) \land X(y) \land X(z) \]
\[ \lor \exists y, z. \quad x = (y \lor \bar{z}) \land X(y) \land \forall w. (Y(w, y) \supset X(z)) \]
\[ \lor \exists y. \quad x = (\check{y}) \land \forall w. X(y(a)) \]
\[ \lor \exists y. \quad x = (\check{y}) \land \forall w. X(y(a)) \]

\[ B \triangleq \lambda x. \quad X(x) \land \]
\[ \lor \exists a, b. \quad x = (a = b) \land a = b \land r = \text{nil} \]
\[ \lor \exists a. \quad x = \hat{N}(a) \land \text{Nat}(a) \land r = \text{nil} \]
\[ \lor \exists y, z. \exists a, b. \quad x = (y \land z) \land r = (a, b) \land Y(a, y) \land Y(b, z) \]
\[ \lor \exists y. \exists a, b. \quad x = (y \lor z) \land r = (a, b) \land a \perp \]
\[ \land (a = \text{true} \supset Y(b, y)) \land (a \neq \text{true} \supset Y(b, z)) \]
\[ \lor \exists y, z. \quad x = (y \lor \bar{z}) \land X(y) \land \forall w. (Y(r(w), y)) \]
\[ \lor \exists y. \quad x = \check{y} \land \forall w. Y(r(w), y(a)) \]
\[ \lor \exists y, z. \exists a, b. \quad x = \check{y} \land r = (a, b) \land Y(a, y(b))) \]

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

Both templates contain negative occurrences of \( Y \). \( \text{BT+HMD}_{\ell} \) allows such occurrences provided \( \text{HMONO}(A, B, X, Y) \) and \( \text{HMONO-Q}(A, B, X, Y) \) 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 \( B \), \( \text{HMONO}(A, B, X, Y) \) \( \text{HMONO-Q}(A, B, X, Y) \) hold.

This lemma is proved in just the same as the case for Prime Numbers. The only negative occurrence of \( Y \) in \( \text{X} \) is in the subformula of the form \( X(x) \land \forall w. (Y(w, y) \supset \cdots) \), so \( A_X[X_0, Y_0] \) is equivalent to \( A_X[X_0, Y_1] \) provided that \( X_0, Y_0 \leq_N (X_1, Y_1) \). 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.
Definition 28

\[
\begin{align*}
\text{Prop} &\triangleq \mu X. (A, B; X, Y) \\
\text{Proves} &\triangleq \mu Y. (A, B; X, Y) \\
\text{True} &\triangleq \lambda x. \exists r. \text{Proves}(r, x)
\end{align*}
\]

The pair \((\text{Prop}, \text{Proves})\) is a provability relation.

We can infer that \(\text{Prop}\) is the least fixpoint of \(A\), while we cannot infer \(\text{Proves}\) is that of \(B\), by the restriction for the \(\mu\)-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 \(\text{BT}+\text{HMID}_0\).

\[\forall x. (\text{True}(x) \supset \text{Prop}(x))\]

However, we need induction only for \(\text{Prop}\) in practice, so \(\mu\)-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 \(\text{nil}\) 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.

\[H \triangleq \lambda x. \ x = \bot\]

\[\lor \exists a. \exists b. \ x = (a=b)\]

\[\lor \exists y. \exists z. \ x = (y \land z) \land X(y) \land X(z)\]

\[\lor \exists y. \exists z. \ x = (y \lor z) \land X(y) \land X(z)\]

\[\lor \exists y. \ x = (\forall y) \land \forall a. X(y(a))\]

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

We can prove the following important properties of Harrop formulas.

Theorem 16

1. \(\text{Harrop}(x) \supset \text{Prop}(x)\)
2. \(\exists f. \exists x. (\text{Harrop}(x) \supset \text{True}(x) \supset \text{Proves}(f, x))\)

4.5. PROVABILITY RELATIONS AND ITS REFINEMENT

These are proved by the induction on \(\text{Harrop}\).

We then define the following new template \(B_1\).

\[B_1 \triangleq \lambda x. B(x, r) \lor \text{Harrop}(x) \land \text{True}(x) \land r = \text{nil}\]

Using \((A, B_1)\), we define a new provability relation.

Definition 29

\[
\begin{align*}
\text{Prop}_1 &\triangleq \mu X.(A, B_1; X, Y) \\
\text{Proves}_1 &\triangleq \mu Y.(A, B_1; X, Y) \\
\text{True}_1 &\triangleq \lambda x. \exists r. \text{Proves}_1(r, x)
\end{align*}
\]

The pair \((A, B_1)\) also satisfies \(\text{HMONO}\) and \(\text{HMONO-Q}\), so we can make use of \(\mu\)-eq-1 and \(\mu\)-ind-1.

We have the following theorem, which shows equivalence of two provability relations.

Theorem 17 We have that

\[\forall x. \text{Prop}(x) \leftrightarrow \text{Prop}_1(x),\]

and

\[\forall x. (\text{Prop}(x) \supset \exists f. \exists x. (\text{Proves}(r, x) \supset \text{Proves}_1(f, r), x))\]

\[\land \exists g. \exists x. (\text{Proves}_1(r, x) \supset \text{Proves}(g, r), x))\]

Proof. (Sketch) Let \(\text{IH}(x)\) (standing for induction hypothesis) be the following formula:

\[
\begin{align*}
\text{Prop}(x) \land \text{Prop}_1(x) \\
\land \exists f. \exists x. (\text{Proves}(r, x) \supset \text{Proves}_1(f, r), x)) \\
\land \exists g. \exists x. (\text{Proves}_1(r, x) \supset \text{Proves}(g, r), x))
\end{align*}
\]

We can prove \(\forall x. \text{Prop}(x) \supset \text{IH}(x)\) and \(\forall x. \text{Prop}_1(x) \supset \text{IH}(x)\) by the induction on \(\text{Prop}\) and \(\text{Prop}_1\). Using Theorem 16, we can always create a candidate proof for a Harrop formula, hence we can calculate \(g\) from the proof of \(\text{Harrop}(x)\).

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.

\[\lambda x. (x = 0 \land A(x)) \lor \exists y. z. (x = (y, z) \land B(x))\]

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 realizor 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 BT + MID, 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 \( \text{HMID}_0 \) 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, \( \text{LTC}_0, \text{LTC}_1, \ldots, \text{LTC}_n \), \( \text{LTC}_0 \) is the basic theory, and \( \text{LTC}_{n+1} \) is a reflected version of \( \text{LTC}_n \). \( \text{LTC}_n \) is the union of each finite level of reflection.

Terms of LTC is essentially combinatory terms with primitive function symbols for natural numbers, lambda terms, and others. Some of them such as 0, Suc, \( \lambda, \vdash \), \( \text{Inl}, \text{Inr} \) are canonical. There are non-canonical function symbols corresponding to these canonical symbols. There are constants which internally represent logical connectives such as \( \lambda, \vdash \) and so on. Atomic formulas are \( \perp, a = b, a \rightarrow b, a \Rightarrow b \). For a typographic reason, we use a different symbol \( \Rightarrow \) from the original paper\(^3\). In LTC, for \( i \geq 1 \), we have two more atomic formulas \( T(a) \) and \( P_i(a) \) 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 \( \Rightarrow \). The formula \( a \Rightarrow b \) means that a term \( a \) evaluates to a canonical form \( b \). Since this relation \( \Rightarrow \) 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 \( \forall' \) and \( \exists' \). The predicate for natural numbers \( \text{Nat}(a) \) can be defined as \( \exists n. (a \Rightarrow n) \).

The logic of \( \text{LTC}_0 \) is an ordinary intuitionistic predicate calculus with equality. Extending \( \text{LTC}_0 \) by reflection, we get \( \text{LTC}_1 \). The rules (called reflection rules) for \( P_i(a) \) and \( T(a) \) are as follows. These rules give connection of internal logic (\( a \) in \( T(a) \)) to external logic (logic of \( \text{LTC}_0 \)).

\[
\Phi^1_i \quad \Phi^1_i \quad \Psi^1_i
\]

where \( \Phi^1_i \) and \( \Psi^1_i \) are defined by the following table.

<table>
<thead>
<tr>
<th>( c )</th>
<th>( \Phi^1_i )</th>
<th>( \Psi^1_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a=b )</td>
<td>( T )</td>
<td>( a=b )</td>
</tr>
<tr>
<td>( a \Rightarrow b )</td>
<td>( T )</td>
<td>( a \Rightarrow b )</td>
</tr>
<tr>
<td>( \perp )</td>
<td>( T )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>( a \land b )</td>
<td>( P_i(a) \land P_i(b) )</td>
<td>( T(a) \land T(b) )</td>
</tr>
<tr>
<td>( a \lor b )</td>
<td>( P_i(a) \lor P_i(b) )</td>
<td>( T(a) \lor T(b) )</td>
</tr>
<tr>
<td>( \forall p )</td>
<td>( \forall x. P_i(p(x)) )</td>
<td>( \forall x. T(p(x)) )</td>
</tr>
<tr>
<td>( \exists p )</td>
<td>( \exists x. P_i(p(x)) )</td>
<td>( \exists x. T(p(x)) )</td>
</tr>
</tbody>
</table>

Next, we give rules for \( \text{LTC}_i \) for \( i > 1 \). In the following, \( \Phi^1_i \) and \( \Psi^1_i \) is in a row of the above table or the following new table.
4.6. INTERPRETATION OF OTHER THEORIES

We take the least fixpoint as \( \mu X. (A_0, B_0; X, Y) \) and \( \mu Y. (A_0, B_0; X, Y) \), and call them \( \text{Prop}_0 \) and \( \text{True}_0 \). 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 \( \text{LTC}_0 \). Then, \( \text{Prop}_0 (F') \) is provable in \( \text{BT+HMID}_0 \), and if \( F \) is provable in \( \text{LTC}_0 \), then \( \text{True}_0 (F') \) is provable in \( \text{BT+HMID}_0 \) where \( F' \) is the translation of \( F \).

Each rule in \( \text{LTC}_0 \) is easily justified. Since \( \text{LTC}_0 \) does not have any induction rules, the restriction of the \( \mu \)-ind-1 rule is not a problem.

We then go to the level 1. We assumed that the language of \( \text{LTC}_0 \) already contained terms like \( \bar{P}_1 (u) \), so the languages of \( \text{LTC}_1 \) are the same, and we continue to use the same definition of \( \text{Term} \) and so on.

We define the templates \( A_1 \) and \( B_1 \) as follows:

\[
A_1 \triangleq \forall x. A_0 (x) \\
B_1 \triangleq \forall x. B_0 (x)
\]

We define \( \text{Prop}_1 \triangleq \mu X. (A_1, B_1; X, Y) \) and \( \text{True}_1 \triangleq \mu Y. (A_1, B_1; X, Y) \). Again, we omit the verification of HMONO and HMONO-Q conditions.

By repeating this kind of construction, we have the translation of \( \text{LTC}_i \) for each \( i \geq 0 \), and \( \text{LTC}_\omega \).
Theorem 19 (Soundness of Translation) Let $F$ be a formula in $\text{LTC}_i$, for $i \geq 0$. Then, $\text{Prop}_i(F')$ is provable in $\text{BT}+\text{HMID}_0$, and if $F$ is provable in $\text{LTC}_i$, then $\text{TRUE}_i(F')$ is provable in $\text{BT}+\text{HMID}_0$ where $F'$ is the $i$-th level translation of $F$.

4.6.3 Reformulating HMID on $\text{LTC}_0$

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 $\text{BT}+\text{HMID}_0$, while our target logic is $\text{LTC}$-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 $\text{LTC}_0$ as our meta-theory, namely we may consider something like $\text{LTC}_0+\text{HMID}$. It seems quite straightforward to move the HMID mechanism onto $\text{LTC}_0$. The resulting theory $\text{LTC}_0+\text{HMID}_0$ would subsume $\text{LTC}_i$ ($i \geq 0$) since we can internally define each $P_i$ and $T_i$ 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 $\text{LTC}_0$. 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 $\text{ITT}$ in our theory.

Since $\text{ITT}$ can be interpreted by $\text{LTC}[3]$, we can indirectly interpret $\text{ITT}$ in our theory by using $\text{LTC}$ 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 $\text{LTC}$: we first define how all the terms in $\text{ITT}$ are translated. Then we 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 $\Sigma$ and $\Pi$ types, but it is treated just the same as in the definition of $A_1$ and $B_1$ for $\text{LTC}$. Finally, we can interpret a first universe in the same way as $A_2$ and $B_2$ for $\text{LTC}$. We can repeat this construction at finitely many times to interpret $\text{ITT}_i$ for each $i \geq 0$.

We also have this interpretation sound: namely if a judgement $J$ is provable in $\text{ITT}_i$, we can prove its translation in $\text{BT}+\text{HMID}_0$.

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 $\text{HMID}$ 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 $\tau$ which takes a term $T$ and a binary predicate $\phi$. Intuitively, $\tau(T, \phi)$ means “$T$ is a type, and $\phi$ is an equivalence relation on the type $T$” (therefore a partial equivalence relation on the whole domain). For some fixed $\tau$, the membership relation $a \in A$ in Martin-Löf’s type theory will be interpreted by $\tau(A, \phi) \wedge \phi(a, a)$ for some $\phi$ where $A$ and $a$ are interpreted by $A$ and $a$. Having this interpretation in mind, $\tau$ 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 $\phi$ is a predicate over individuals (terms), $\tau$ is a second-order predicate, hence the whole theory becomes impredicative. Smith[40] formalized Allen’s interpretation in $\text{CTT}$, 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 $\text{HMNO-Q}$ 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, $\text{Proves}(a, A)$ 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 \( \phi^* \) where \( \tau \) 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 \((\text{Prop}, \text{Proves})\). In some respects, his method is more flexible than ours, since he allows arbitrary recursive function for describing the second predicate \((x \in \text{Proves}(x, y))\) 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 \( \text{Prop}(p) \) and \( \text{Proves}(a, p) \). If we introduce an IMH/L-like mechanism into \( \mathcal{RPT} \), we would be able to define \((\text{Prop}, \text{Proves})\) 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.

---

Chapter 5

Conservativeness of \( \Lambda! \) over \( \lambda\sigma \)-calculus

\( \Lambda! \) is a unique functional programming language which has the facility of the encapsulated assignment, without losing referential transparency[34]. The \texttt{let}-construct in \( \Lambda! \) can be considered as an environment, which has a close relationship to the substitution in the calculus for explicit substitution \( \lambda\sigma \)-calculus in [1].

This chapter discusses the relationship between these two calculi. We first define a slightly modified version of \( \Lambda! \) which adopts de Bruijn’s index notation. We then define an injective map from \( \lambda\sigma \)-calculus to \( \Lambda! \), and show that the Beta-reduction and the \( \sigma \)-reductions in \( \lambda\sigma \)-calculus correspond to the \( \beta \)-reduction and \texttt{let}-reductions in \( \Lambda! \), respectively. Finally, we prove that, as equality theories, \( \Lambda! \) is conservative over \( \lambda\sigma \)-calculus.

5.1 Introduction

\( \Lambda! \) 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, \( \Lambda! \) programs can be quite efficient compared with programs written in ordinary functional programming languages. In spite of the existence of assignment, \( \Lambda! \) 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 \( \Lambda! \) is a good starting point of treating assignment in a mathematically well-founded manner.

In \( \Lambda! \), the \texttt{let}-construct plays a fundamental role. The evaluation of the \texttt{let}-construct \((\texttt{let} \ (x \ a) \ b)\) can be naturally considered as evaluating \( b \) under the environment \( x = a \). This concept of environment is closely related to substitution in \( \lambda\sigma \)-calculus[1]. \( \lambda\sigma \)-calculus is an extension of \( \lambda \)-calculus where substitution has its own syntax, and explicitly described. \( \lambda\sigma \)-calculus is mathematically well founded, since it is conservative over \( \lambda \)-calculus.
This chapter discusses the relationship between ΛT and λσ-calculus. First, we define a slightly modified version of ΛT. The version we present in this chapter adopts de Bruijn's index notation, and has a slightly extended λt-reductions compared with the original definition given by Satō [36]. Next, we define an injective map Ω from λσ-calculus to ΛT. Then, we show that the Beta-reduction and the σ-reductions in λσ-calculus correspond to the β-reduction and λt-reductions in ΛT. Finally, we prove that, as equality theories, ΛT is conservative over λσ-calculus. Namely, we have that s = t in λσ-calculus if and only if Ω(t) = Ω(t) in ΛT.

In the following, we use metavariables t, s, u for λσ-terms, β, φ, χ for λσ-substitutions, a, b, c for ΛT-terms, n, m for natural numbers.

5.2 λσ-calculus with de Bruijn index

We quote the untyped λσ-calculus in de Bruijn’s index notation from [1]. We assume that readers are familiar with de Bruijn's notation and λσ-calculus. See also [1] and [13].

Definition 30 (Term t and Substitution θ)

\[ t ::= \text{id} \mid t u \mid \lambda t \mid t[θ] \]
\[ θ ::= \text{id} \mid \uparrow \mid t \cdot θ \mid θ φ \]

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 \( λx.λy.xy \) will be represented by \( λ(λ(21)) \) 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 \( t[θ] \) is the term \( t \) to which the substitution \( θ \) is applied.

Each substitution intuitively represents a simultaneous substitution for indices. The substitution \( \text{id} \) is the identity substitution. The substitution \( \uparrow \) is the “shift” operator, which substitutes \( n + 1 \) for each index \( n \). The substitution \( t \cdot θ \) is “cons” of a term \( t \) and a substitution \( θ \), which intuitively represents the substitution \( \{1 := t_1, 2 := s_2, \cdots \} \) where \( t \) means \( \{1 := t_1, 2 := s_2, \cdots \} \). Finally, \( θ φ \) is the composition of two substitutions.

Definition 31 (Context C)

\[ C ::= () \mid Ct \mid tC \mid λC \mid C[θ] \mid t[θ] \]
\[ θ ::= () \mid C \cdot θ \mid t \cdot θ \mid θ φ \mid θ φ θ \]

5.2 λσ-CALCULUS WITH DE BRUIJN INDEX

A context \( C \) has just one hole \( () \). To emphasize it, we sometimes use the notation \( C(\)\). We may replace the hole with a term \( t \) or a substitution \( θ \) in a context \( C(\)\), which is denoted as \( C(t) \) or \( C(θ) \).

Definition 32 (1-step reduction \( → \))

\[ \text{Beta} \quad (λt)s → t[s \cdot \text{id}] \]
\[ \text{Var}_1D \quad 1[\text{id}] → 1 \]
\[ \text{Var}_CCons \quad 1(t \cdot θ) → t \]
\[ \text{App} \quad (t_1)[θ] → (t[θ])(s[θ]) \]
\[ \text{Abs} \quad (λt)[θ] → λ(t[θ]) \]
\[ \text{Clos} \quad t[θ][c] → t[θ φ] \]
\[ \text{IdL} \quad \text{id} \circ θ → θ \]
\[ \text{ShiftId} \quad \uparrow \circ \text{id} \uparrow \]
\[ \text{ShiftCons} \quad t \cdot θ → θ \]
\[ \text{Map} \quad (t \cdot θ)c χ → t[θ](t[θ φ]) \]
\[ \text{Ass} \quad θ φ \circ θ → θ φ \circ φ \]

Rules other than \text{Beta} are called \( σ \)-rules or σ-reductions. Reduction relations for the \text{Beta}-rule and the \( σ \)-rules are written as \( →_β \) and \( →_σ \). \text{Beta}-reduction corresponds to the usual β-reduction in λ-calculus, but it does not actually perform the substitution. It merely adds a new substitution \( s \cdot \text{id} \) to the term \( t \). This substitution will be later resolved by \( σ \)-reductions.

Definition 33 (Reduction \( → )\) The relation \( → \) is the least relation satisfying the following conditions:

1. \( → \) is reflexive and transitive.
2. \( t → s \) implies \( C(t) → C(s) \).
3. \( θ → φ \) implies \( D[θ] → D[φ] \).

The equality = is the equivalence relation induced by \( → \).

Theorem 20 (Abadi et al) The σ-reduction is confluent and terminating. The λσ-calculus is confluent.

The σ-normal form of a λσ-term \( t \) is the normal form of \( t \) under the σ-reductions, and is written as \( σ(t) \).
5.3 \( \Lambda \) and plet-calculus

5.3.1 A Functional Programming Language \( \Lambda \)

\( \Lambda \) 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, \( \Lambda \) does not lose contextually 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 \( \Lambda \) is a conservative extension of \( \lambda \)-calculus. The terms in \( \lambda \)-calculus can be naturally translated into \( \Lambda \), however, it is not clear that the equality is preserved through this translation, since the introduction of assignment to \( \Lambda \) forces us to fix evaluation order to some extent while \( \lambda \)-calculus allows strong reductions, which may reduce subterms inside \( \lambda \) in an arbitrary context. Therefore, conservativeness of \( \Lambda \) over \( \lambda \)-calculus is an interesting problem.

5.3.2 Modification to \( \Lambda \)

This subsection explains the two different points between the original version of \( \Lambda \) 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]):

\[
(\text{let} \ ((x \ t)) \ (\text{apply} \ a \ b)) = (\text{apply} \ (\text{let} \ ((x \ t)) \ a) \ (\text{let} \ ((x \ t)) \ b))
\]

In this example, \( t, a, \) and \( b \) represent some terms in \( \Lambda \). In a natural translation from \( \lambda \)-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 \( \Lambda \)):

\[
(\text{lambda} \ (y) \ (\text{let} \ ((x \ y)) \ (\text{apply} \ a \ b))) = (\text{lambda} \ (y) \ (\text{apply} \ (\text{let} \ ((x \ y)) \ a) \ (\text{let} \ ((x \ y)) \ b)))
\]

In the original \( \Lambda \), we have no way to evaluate the subterm \( (\text{let} \ ((x \ y)) \ (\text{apply} \ a \ b)) \), since \( y \) is not closed. From the Church-Rosser Property of the original \( \Lambda \), we can show that the equation above does not hold, which means the original version is not conservative over \( \lambda \)-calculus under a natural translation. It follows that the original \( \Lambda \) is not conservative over the pure \( \lambda \)-calculus; two equal \( \lambda \)-terms \( \lambda y.((\lambda x.x)y) \) and \( \lambda y. y \) are translated into two \( \Lambda \)-terms which are not equal.

This example motivates our modification. We allow reductions of a term \( (\text{let} \ ((x \ t)) \ a) \), 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 \( \text{let-construct or lambda-construct in } a \). By extending \( \text{let-reduction in this way, we can reduce, for example, the term like:}

\[
(\text{lambda} \ (y) \ (\text{let} \ ((x \ y)) \ (\text{apply} \ a \ b))) \rightarrow (\text{lambda} \ (y) \ (\text{apply} \ a \ (y \ b)))
\]

where \( a \ N \) 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 \( \Lambda \), and use the term the “original” version if we mention \( \Lambda \) in [34].

5.3.3 Definition of \( \Lambda \)

The set of \( N \)-terms is defined for each natural number \( N \) as follows:

**Definition 34 (Term \( a_N \) of \( \Lambda \))**

\[
a_N \ := \ n \ if \ N \geq 1
\]

\[
| \ (\text{set} \ n \ a_N) \ if \ N \leq n
\]

\[
| \ (\text{let} \ ((a_N)) \ a_{N+1}
\]

\[
| \ (\text{while} \ a_N \ a_N \ a_N)
\]

\[
| \ (\text{if} \ a_N \ a_N \ a_N)
\]

\[
| \ \text{nil}
\]

\[
| \ (\text{null} \ a_N)
\]

\[
| \ (\text{pair} \ a_N \ a_N)
\]

\[
| \ (\text{pair} \ a_N)
\]

\[
| \ (\text{car} \ a_N)
\]

\[
| \ (\text{cdr} \ a_N)
\]

\[
| \ (\text{lambda} \ () \ a_1)
\]

\[
| \ (\text{fun} \ a_N)
\]

\[
| \ (\text{apply} \ a_N \ a_N)
\]

\[
| \ (\text{mu} \ a_N)
\]

1It follows that Theorem 4.6 in [34] also needs the modification to the definition of \( \Lambda \).
CHAPTER 5. CONSERVATIVENESS OF $\Lambda^!$

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 $(\text{set} \ n \ a_N)$ 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 $(\text{null} ? \ a)$ are predicates which decide whether $a$ is nil or not, and return true or false. The term $(\mu a)$ is the p-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, $(\text{pair} \ a_N \ b_N)$, and $(\lambda () a_N)$ are called constructor terms, and the terms $(\text{null} ? \ a_N)$, $(\text{pair} ? \ a_N)$, and $(\text{fun} ? \ a_N)$ are called recognizer terms. We also say that nil and (null? $a_N$) are of the same kind. Likewise, $(\text{pair} \ a_N \ b_N)$ and $(\text{pair} ? \ a_N)$, are of the same kind, and $(\lambda () a_N)$ and $(\text{fun} ? \ a_N)$ are of the same kind. Other combinations of pairs of these terms are of different kinds.

The reduction rules of $\Lambda^!$ are listed in the Appendix of this chapter. The confluence of the original $\Lambda^!$ was proved in [34], and the confluence of this modified version is proved similarly. The equality in $\Lambda^!$ is the let 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 $\Lambda^!$.

Example 1 (Reduction in $\Lambda^!$) Let $t$ be the following term.

\[ (\lambda () \text{(apply} (\lambda () (\lambda () (\text{pair} 1 (\text{pair} 2 3)))) \ 1)) \text{nil}) \]

If we use the notation with variable names, $t$ is written as follows:

\[ (\lambda (x) \text{(apply} (\lambda (y) (\lambda (z) (\text{pair} z (\text{pair} y x)))) x)) \text{nil}) \]

The following sequence is a reduction sequence starting from $t$.

\[ t \rightarrow (\text{by Rule 12}) \]

\[ (\lambda () \text{nil}) \]

In this chapter, we are mainly concerned with the fragment of $\Lambda^!$ 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)

\[ a := n \text{ if } n \geq 1 \]

\[ (\text{let} ((a_1) (a_2) \cdots (a_k)) b) \]

\[ (\lambda () a) \]

\[ (\text{apply} a b) \]

Since plet-calculus is solely used for the translation, we do not define reduction rules for it.
Chapter 5. Conservativeness of $\Lambda$

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:

$$
\delta(n) \triangleq n \text{ if } n \geq 1
$$

$$
\delta((\text{let } ((a_1) \ldots (a_k)) \ b)) \triangleq \max(\delta(a_1), \ldots, \delta(a_k), \delta(b) - k)
$$

$$
\delta((\text{lambda } () a)) \triangleq \max(1, \delta(a) - 1)
$$

$$
\delta((\text{apply } a \ b)) \triangleq \max(\delta(a), \delta(b))
$$

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:

$$
t^\dagger \triangleq 1
$$

$$
(ta)^\dagger \triangleq (\text{apply } t^\dagger \ s^\dagger)
$$

$$
(\lambda t)^\dagger \triangleq (\text{lambda } () t^\dagger)
$$

$$
(i[\theta])^\dagger \triangleq (\text{let } \theta^{\delta(t^\dagger)} t^\dagger)
$$

The translation for substitution $-^{(n)}$ is defined as follows.

Definition 38 (Translation $\theta^{(n)}$) For a substitution $\theta$ 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:

$$
id^{(n)} \triangleq (((1) \ldots (n)))
$$

$$
t^\dagger\theta^{(n)} \triangleq (((2) \ldots (n + 1)))
$$

$$
(a \cdot \theta)^{(n)} \triangleq (((a^\dagger) (b_1) (b_2) \ldots (b_k) (k + 2))
$$

if $\theta^{(n)}$ is $((b_1) (b_2) \ldots (b_k))$

$$
(\theta \circ \phi)^{(n)} \triangleq (((\text{let } \theta^{(m)} (a_1)) \ldots (\text{let } \theta^{(m)} (a_k)))
$$

if $\theta^{(n)}$ is $((a_1) (a_2) \ldots (a_k))$

and $m$ is $\max(\delta(a_1), \ldots, \delta(a_k))$.

Proposition 3 The translation $-^1$ is injective.

Proof. First note that, for each substitution $\theta$ 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 $\text{id}^{(n)}$ nor $\uparrow^{(n)}$. Moreover, its last element is a natural number $k + 2$, and differs\(^2\) from the last element of $\theta^{(n)}$. 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 $\theta$. $\square$

5.4.2 Translation from plet-calculus to $\Lambda$!

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,

$$(\text{apply } 3 \text{ (lambda } () \text{ (apply } 1 \text{ 2)))}^+$$

is $$(\text{apply } 4 \text{ (lambda } () \text{ (apply } 1 \text{ 3)))}$$. 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 ($a^+_m$)

$$
n^+_m \triangleq n \text{ if } n \leq m
$$

$$
n^+_m \triangleq n + 1 \text{ if } n > m
$$

$$
((\text{let } ((a_1) \ldots (a_k)) \ b)^+ \triangleq (\text{let } ((a_1^+_m) \ldots (a_k^+_m)) \ b)
$$

$$
(\text{lambda } () a)^+ \triangleq (\text{lambda } () a^+_m)
$$

$$
(\text{apply } a \ b)^+ \triangleq (\text{apply } a^+_m \ b^+_m)
$$

We simply write $a^+_m$ as $a^+$.

Definition 40 (Translation * from plet-calculus to $\Lambda$!)

$$
n^* \triangleq n
$$

$$
((\text{let } () \ b)^* \triangleq (\text{let } (b) 1)
$$

$$
((\text{let } ((a_1) \ldots (a_k)) \ b)^* \triangleq (\text{let } ((a_1) \ldots (a_k)) \ c)
$$

if $c$ is $((\text{let } (a_1^+ \ldots (a_k^+)) \ b)^*$

$$
(\text{lambda } () a)^* \triangleq (\text{lambda } () a^+)
$$

$$
(\text{apply } a \ b)^* \triangleq (\text{apply } a^+ \ b^+)
$$

\(^2\)This is the reason why we attached the (meaningless) term $k + 2$ in the definition.
In the following, we sometimes regard \texttt{let}-terms as pure terms in \(\Lambda^1\) (through the translation \(\ast\)).

The translation \(\Phi\) from \(\lambda\sigma\)-calculus to the pure-fragment is defined as follows.

**Definition 41 (Translation \(\Phi\) from \(\lambda\sigma\)-calculus to \(\Lambda^1\))**
\[
\Phi(t) \triangleq (t^\ast)^\ast
\]

**Theorem 21** \(-^*\) is injective. Hence \(\Phi\) is injective.

**Proof.** Clear.

**Remark 3** If a non-injective map were allowed as the translation \(\Phi\), part of our results (the first part of Theorem 22) would become trivial as shown below.

In \(\lambda\sigma\)-calculus, the set of \(\sigma\)-normal forms can be regarded as the set of pure \(\lambda\)-terms, so the map \(\sigma(\_\_\) can be regarded as the translation from \(\lambda\sigma\)-terms to pure \(\lambda\)-terms. We have that, \(s = t\) holds if and only if \(\sigma(s) = \sigma(t)\) holds. \(\Lambda^1\) is also conservative over pure \(\lambda\)-calculus. Namely, there is a map \(\Psi\) from pure \(\lambda\)-terms to \(\Lambda^1\)-terms such that \(a = b\) holds if and only if \(\Psi(a) = \Psi(b)\) holds. Let \(\Phi\) be the composition of \(\sigma\) and \(\Psi\); then we have that \(s = t\) holds in \(\lambda\sigma\)-calculus if and only if \(\Phi(s) = \Phi(t)\) holds in \(\Lambda^1\). \(\Box\)

### 5.4.3 Properties of the translation \(\Phi\)

Here, we prove that the translation \(\Phi\) 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 \((\text{let} (a) \ b)\) reduces to \(b[1 := a, 2 := 1, 3 := 2, \ldots, k := k - 1]\) using \textit{let}-rules only, where \(k\) is \(\delta(b)\).

Here \([1 := a, 2 := 1, 3 := 2, \ldots, k := k - 1]\) denotes the simultaneous substitution. Note that \(a\) and \(b\) are not necessarily \(\Omega\)-terms. As was stated in Section 5.3.2, this lemma does not hold for the original \(\Lambda^1\), since we cannot reduce the term \((\text{let} (a) b)\) 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 \((\text{let} (a) b)\).

Similarly, we have the following lemma.

**Lemma 9** Let \(a_1, \ldots, a_k\) and \(b\) be pure \(N\)-terms, and \(k\) be \(\delta(b)\). If \(n \geq k\), then \((\text{let} (a_1) \ldots (a_k) b)\) reduces to \(b[1 := a_1, 2 := a_2, \ldots, k := a_k]\) using \textit{let}-rules only.

\(\ast\)This claim does not hold for the original \(\Lambda^1\), but it does hold for the modified version presented in this chapter.

5.4. **Translation of \(\lambda\sigma\)-CALCULUS**

Note that, we regard \texttt{let}-terms as \(\Lambda^1\)-terms through the translation \(\ast\) 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 \(\Lambda^1\). For each substitution-reduction rule \(\theta \rightarrow \phi\) and a term \(s\) in \(\lambda\sigma\)-calculus, \(\Phi(s[\theta]) = \Phi(s[\phi])\) holds in \(\Lambda^1\).

**Proof.** This proposition is proved by the case-analysis.

**Beta** The left hand side (LHS, in short) of \(\textit{Beta}\) is translated into

\[
(\text{apply} (\text{lambda} () t^\ast) s^\ast)
\]

which \(\beta\)-reduces to

\[
(\text{let} (\{s^\ast\}) t^\ast)
\]

By Lemma 8, this is equal to \(t^\ast[1 := s^\ast, 2 := 1, 3 := 2, \ldots]\).

The right hand side (RHS, in short) of \(\textit{Beta}\) is translated into

\[
(\text{let} (\{s^\ast\} (1) \cdots (n - 1)) t^\ast)
\]

Calculation of indexes shows that this is equal to \(t^\ast[1 := s^\ast, 2 := 1, 3 := 2, \ldots]\).

**VarID** LHS is translated into \((\text{let} (\{1\}) 1\) which reduces to 1. RHS is translated into 1.

**VarCons** LHS is translated into \((\text{let} ((t^\ast) \cdots) 1\) which reduces to \(t^\ast\). RHS is translated into \(t^\ast\).

**App** Suppose \(\theta((t^\ast))\) is \((a_1) \ldots (a_k))\).

LHS is translated into \((\text{let} ((a_1) (a_2) \cdots (a_k)) (\text{apply} t^\ast s^\ast))\).

RHS is translated into

\[
(\text{apply}
\begin{align*}
&((\text{let} ((a_1) (a_2) \cdots (a_k)) t^\ast)) \\
&((\text{let} ((a_1) (a_2) \cdots (u_m)) s^\ast))
\end{align*}
\]

where \(l\) and \(m\) are \(\delta(t^\ast)\) and \(\delta(s^\ast)\).

By Lemma 9, \((\text{let} (a_1) \cdots (a_k)) t^\ast\) is equal to
\[
\text{LHS} \text{ and RHS reduce to }
\]

\[
\text{(Abs)} \text{ Suppose } \theta^{(\ell_1)} \text{ is } (a_1) \cdots (a_k). \\
\text{LHS is translated into } \left(\text{let } ((a_1) \cdots (a_k)) \text{ (lambda } \ell_1)\right). \text{ This reduces to }
\]

\[
\text{RHS is translated into } \\
\text{(lambda) } \\
\left(\text{let } ((1) \text{ (2) } \cdots a_1)) \cdots ((\text{let } ((2) (3) \cdots a_k))) \ell_1)\right)
\]

The latter term reduces to the former by Lemma 9.

\[
\text{(Clos)} \text{ Suppose } \theta^{(t_1)} \text{ is } (a_1) \cdots (a_n), \text{ and } \phi^{(n)} \text{ is } (b_1) \cdots (b_m) \text{ where } n \text{ is } \max(a_1, \cdots, a_l). \\
\text{LHS is translated into } \\
\left(\text{let } ((b_1) \cdots (b_n)) \text{ (let } (a_1) \cdots (a_k) \ell_1)\right)
\]

RHS is translated into

\[
\left(\text{let } ((\text{let } ((b_1) \cdots (b_n)) a_1)) \cdots ((\text{let } ((b_1) \cdots (b_n)) a_k)) \ell_1)\right)
\]

LHS and RHS reduce to

\[
\text{5.4. TRANSLATION OF } \lambda \sigma\text{-CALCULUS }
\]

\[
\text{apply } \ell_1' \{1:a_1, \cdots, l:a_l\}
\]

\[
\text{and similarly for } s'. \text{ Hence, by Lemma 8, LHS and RHS reduce to }
\]

\[
\text{(IdL) Let } n \text{ be } \ell(s). \text{ Suppose } \theta^{(n)} \text{ is } (b_1) \cdots (b_k). \text{ Then, } s[\ell \circ \theta] \text{ is translated into }
\]

\[
\text{let } ((\text{let } (b_1) \cdots (b_k) \ell_1)) \cdots ((\text{let } (b_1) \cdots (b_k) n)))
\]

\[
s_1'
\]

Since } k \geq n, \text{ this reduces to } \left(\text{let } ((b_1) \cdots (b_n)) s') \text{ Then this term is identical to } \left(\text{let } s[\ell \circ \theta] \right) \text{ by Lemma 9.}

\[
\text{(ShiftId) Let } n \text{ be } \ell(s). \text{ The term } s[\ell \circ \text{id}] \text{ is translated into }
\]

\[
\left(\text{let } ((\text{let } (1) \cdots (n+1) 2)) \text{ (let } (1) \cdots (n+1) 3) \cdots ((\text{let } (1) \cdots (n+1) n+1)))
\]

\[
s_1'
\]

This reduces to } \left(\text{let } ((2) \cdots (n+1)) s') \text{ which is identical to } \left(\text{let } s[\ell] \right)'.

\[
\text{(ShiftCons) Let } n \text{ be } \ell(s). \text{ Suppose } \theta^{(n+1)} \text{ is } (b_1) \cdots (b_k). \text{ Then, } s[\ell \circ (t \circ \theta)] \text{ is translated into }
\]

\[
\left(\text{let } ((\text{let } (t_1') (b_1) \cdots (b_k) k+2) 2)) \text{ (let } (t_1') (b_1) \cdots (b_k) k+2) 3) \cdots ((\text{let } (t_1') (b_1) \cdots (b_k) k+2) n+1)))
\]

\[
s_1'
\]

We also have } k \geq n+1, \text{ and the term above reduces to }

\[
\left(\text{let } (b_1) \cdots (b_n) s')
\]

which is equal to } \left(\text{let } s[\ell_1] \right)'.

\[
\text{(Map) Let } n \text{ be } \ell(s). \text{ Suppose } \theta^{(n)} \text{ is } (a_1) \cdots (a_k), \text{ and } \chi^{(m)} \text{ is } (b_1) \cdots (b_l) \text{ where } m \text{ is } \max(\delta(t), \delta(a_1), \cdots, \delta(a_k)) \text{ where } k+2.
\]

\[
\text{Then, } s[t \circ \theta \circ \chi] \text{ is translated into }
\]

\[
\text{let } (b_1) \cdots (b_n) s')
\]
CHAPTER 5. CONSERVATIVENESS OF $\Lambda^!$

(1let (((1let ((b_1) ... (b_n)) t'))) 
(1let ((b_1) ... (b_n) a_i)) 
... 
(1let ((b_1) ... (b_n) a_k)) 
(1let ((b_1) ... (b_n) k + 2))) 

$s'$

$s[\chi o (\theta o \chi)]$ is translated into

(1let (((1let ((b_1) ... (b_n)) t'))) 
(1let ((b_1) ... (b_n) a_i)) 
... 
(1let ((b_1) ... (b_n) a_k)) 
(k + 2)) 

$s'$

where $\delta(t)$ is p, $\chi^{(p)}$ is ((b_1) ... (b_n)), max($\delta(a_1)$, ..., $\delta(a_k)$) is r, and $\chi^{(r)}$ is ((b_1) ... (b_n)).

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

\[
\text{let}\{1:=a_1|1:=b_1|1:=c_1,\ldots, m:=c_m|1:=b_1,\ldots, m:=c_m\}, \\
\text{let}\{1:=a_1|1:=b_1|1:=c_1,\ldots, m:=c_m|1:=b_1,\ldots, m:=c_m\}
\]

Note that, we have used only let-rules for proving the cases for $\sigma$-rules. Note also that, the 1-step Beta-reduction can be simulated by the 1-step $\beta$-reduction with some let-rules.

**Proposition 5** Let \( t \) and \( s \) be $\lambda\sigma$-terms. If \( t = s \), then \( \Phi(t) = \Phi(s) \) in $\Lambda^!$.

**Proof.** We first prove that, the result of Proposition 4 can be extended to an arbitrary context. Namely, for a context $C()$, if \( t \rightarrow s \), then \( \Phi(C(t)) = \Phi(C(s)) \) where \( t \) and \( s \) are terms or substitutions. These are straightforward if the used reduction rule is a $\sigma$-rule. However, in the case of Beta-rule, there occurs a subtle point; for example, \((\lambda x(2)3)[1] \) is translated into

\[
\text{let}\{(2)(3)(4)\} (\text{apply} (\lambda x x) 2 3).
\]

On the other hand, the result of applying Beta rule to it is \( 1[1] \) which is translated into \( \text{let}\{(2)\} 1 \). We can use Lemma 9 to overcome this difficulty, and can prove the equality of $\Phi(C(t))$ and $\Phi(C(s))$.

Finally, we can extend the result for 1-step reductions to the general case, and get the desired proposition. \( \square \)

By checking the proofs, we know that, if \( t = s \) is shown by $\sigma$-rules only, then \( \Phi(t) = \Phi(s) \) is shown by let-rules only.

### 5.5 Translation of the pure-fragment into $\lambda\sigma$-calculus

We now define the reverse translation, namely the translation from the pure-fragment of $\Lambda^!$ to $\lambda\sigma$-calculus.


**Definition 42 (Translation \( \Psi \))**

\[
\Psi(n) \triangleq 1[\uparrow^n] \\
\Psi((\text{let } \((a)\) \ b)) \triangleq \Psi(b)[\Psi(a) \cdot \text{id}] \\
\Psi((\lambda a)(a)) \triangleq \lambda a \Psi(a) \\
\Psi((\text{apply } a \ b)) \triangleq \Psi(a)\Psi(b)
\]

In the first clause, \(1[\uparrow^n] \) is \(n\)-times application of substitution, that is, \(1[\uparrow^n] = \underbrace{1 \cdot 1 \cdots 1}_{n \text{ times}} \).

**Proposition 6** We have the following:

1. \( \Psi \) is injective.
2. Let \( a \) and \( b \) be pure \( \Lambda \)-terms in \( \Lambda_\ast \). If \( a = b \), then \( \Psi(a) = \Psi(b) \) holds.

\textbf{Proof.} We first prove the theorem for the case of \( a \rightarrow b \). It is proved by the induction on the derivation of \( a \rightarrow b \). 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 \((\text{apply } \lambda c \ b) \), \( b \) be \((\text{let } ((d')) c') \), \( c \rightarrow c' \) and \( d' \rightarrow d' \). Then, \( \Psi(a) = \lambda c \Psi(c) \Psi(d) \), and \( \Psi(b) = \Psi(c') \Psi(d') \cdot \text{id} \).

By the induction hypothesis and the Beta rule in \( \Lambda \)-calculus, these terms are equal.

**(Rule 16)** Suppose \( a \) is \((\text{let } ((c)) d) \), \( d \rightarrow e \), and \( b = e^{-} \). We have \( \Psi(a) = \Psi(d) [\Psi(c) \cdot \text{id}] \). We can show that, all the occurrences of 1 in \( \Psi(d) \) are followed by one or more \( \uparrow \)'s, hence \( \Psi(d) [\Psi(c) \cdot \text{id}] \) is equal to \( \Psi(d)[1 \cdot 1 \cdot 1 \cdots 1] \), and \( \Psi(e^\rightarrow) \cdot \Psi(c) \cdot \text{id} \), which is again equal to \( \Psi(d) \). We have \( d \not\in FV(c) \), and therefore, \( d' \rightarrow e' \). By the induction hypothesis \( \Psi(d') = \Psi(e') \), hence \( \Psi(a) = \Psi(b) \).

**(Rule 17)** Suppose \( a \) is \((\text{let } ((c)) d) \), \( \rho(d, p) = 1, c \rightarrow c', d \rightarrow d' \), and \( c \equiv d'[c''] \) is \((\text{let } ((c')) d') \). Then, \( \Psi(a) = \Psi(d) [\Psi(c') \cdot \text{id}] \), which is equal to \( \Psi(d') [\Psi(c') \cdot \text{id}] \), by the induction hypothesis. By the induction on the term \( \Psi(d') \), we have that the \( \sigma \)-normal forms of this term and the term \( \Psi(e) [\Psi(c') \cdot \text{id}] \) are identical. Hence we have that \( \Psi(a) \) and \( \Psi(b) \) are \( \sigma \)-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 \( \lambda \sigma \)-term \( t \), \( \Psi(\Phi(t)) = t \) holds. Moreover, the equality is shown by the \( \text{let} \)-rules only.

\textbf{Proof.} This proposition is proved by the induction on the term \( t \). □

**Theorem 22** Let \( t \) and \( s \) be \( \lambda \sigma \)-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 \( \sigma \)-rules only, \( \Phi(t) \) and \( \Phi(s) \) are shown to be equal using \( \text{let} \)-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 \( \beta \)-rule, and some applications of \( \text{let} \)-rules.

\textbf{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 \( \Lambda_\ast \) and \( \lambda \sigma \)-calculus have a close relationship; as equality theories, \( \Lambda_\ast \) (the version presented in this chapter) is conservative over \( \lambda \sigma \)-calculus.

However, we can see several differences between them. Firstly, the reduction rules do not directly correspond, namely, \( t \rightarrow s \) in \( \lambda \sigma \)-calculus does not necessarily imply \( \Phi(t) \rightarrow \Phi(s) \) in \( \Lambda_\ast \). Secondly, substitutions are objects in \( \lambda \sigma \)-calculus, and can be directly treated, while its corresponding expression \((\text{let } (((c)) d)) \) is not a term in \( \Lambda_\ast \). This reflects that, in \( \Lambda_\ast \) we only consider the environment with some term, and never treat one as an independent object. One of the design goals of \( \Lambda_\ast \) 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 \( \lambda \sigma \)-calculus, we have a complete normal-order strategy for the reductions in \( \Lambda_\ast \), which we plan to implement on a computer. □

**5.7 Conclusion**

We have shown the rigid relationship between "explicit substitution" (\( \lambda \sigma \)-calculus) and our functional language \( \Lambda_\ast \). We first presented a modified version of \( \Lambda_\ast \) so that we may reduce \( \text{let} \)-terms under the "read-only" condition. We used de Bruijn's index notation in this presentation. We then gave a translation from \( \lambda \sigma \)-calculus into the pure-fragment of \( \Lambda_\ast \), and a reverse one. We proved that, through these translations, \( \sigma \)-rules correspond to \( \text{let} \)-rules, Beta-rule corresponds to \( \beta \)-rule, and finally \( \Lambda_\ast \) is conservative over \( \lambda \sigma \)-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 \( \Lambda^1 \) is a well-founded programming language with assignment.

As a future work, we should extend RPT so that reasoning about \( \Lambda^1 \)-programs can be formalized in RPT, and then extend our Constructive Programming System to include such reasoning. By doing these things, we can synthesize \( \Lambda^1 \)-programs (with the assignment and the while statements) by the way of Constructive Programming.

Appendix: The Definition of \( \Lambda^1 \) in de Bruijn's notation

The Appendix gives several definitions including the reduction rules of \( \Lambda^1 \) in de Bruijn's index notation.

A position is a finite sequence of positive integers, with \( \epsilon \) 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,

\[
\text{(apply (apply } a \; b) \; c) /22 \text{ is } a, \text{ and} \\
\text{(apply (apply } a \; b) \; c)/\epsilon \text{ is (apply (apply } a \; b) \; c).
\]

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**

\[
\nu(\text{let } (b) \; c, 211 q) \triangleq \nu(b, q) \\
\nu(\text{let } (\text{let } (b) \; c, 3 q) \triangleq \nu(c, q) + 1 \\
\nu(\text{lambda } (b) \; c, 3 q) \triangleq \nu(b, q) + 1 \\
\nu((f \; b_1 \cdots b_n), q) \triangleq \nu(b_{n-1}, q) \text{ if } f \text{ is not let nor lambda,} \\
\nu(a, p) \triangleq 0 \text{ otherwise}
\]

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.

\( \text{FV}(a) \) is the set of \( \rho(a, p) \) where \( p \) ranges over all the free occurrences of variables in \( a \).

5.7. CONCLUSION

Let us take an example. Let \( a \) be \( (\text{let } ((2)) \; (\text{pair } 1 \; 2)) \). Then \( \nu(a, 211) = 0 \), the occurrence at 211 of \( a \) is free, and \( \rho(a, 211) = 2 \). \( \nu(a, 32) = 1 \), the occurrence at 32 of \( a \) is bound, and \( \rho(a, 32) \) is undefined. \( \nu(a, 33) = 1 \), the occurrence at 33 of \( a \) is free, and \( \rho(a, 33) = 1 \).

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,

\[
a^+ \text{ is } (\text{let } ((3)) \; (\text{pair } 1 \; 3)) \\
a^- \text{ is } (\text{let } ((1)) \; (\text{pair } 1 \; 1))
\]

A precise definition of \( a^+ \) is given by Definition 39.

An \( N \)-term \( a \) is called \( N \)-closed if \( \text{FV}(a) \cap [1, \ldots, N] = \emptyset \). 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 \( (\text{set } \; n \; b) \), \( n \) 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 \( p = \epsilon \), \( a_p[d] \) is \( d \).
- Otherwise,

\[
\nu(a, p) \triangleq \begin{cases} 
\text{if } a \text{ is } (\text{lambda } (b) \; c), \\
\text{then } a_p[d] \text{ is } (\text{lambda } (b \; [d^+] \; c)) \text{ if } p \text{ is } 3q, \text{ and is undefined otherwise.} \\
\text{if } a \text{ is } (\text{let } ((b)) \; c), \\
\text{then } a_p[d] \text{ is } (\text{let } ((b \; [d^+] \; c)) \text{ if } p \text{ is } 211q, a_p[d] \text{ is } (\text{let } ((b) \; c) \; [d^+] \text{ if } p \text{ is } 3q, a_p[d] \text{ is undefined otherwise.} \\
\text{if } a \text{ is } (f \; b_1 \cdots b_n), \\
\text{where } f \text{ is not let nor lambda,} \\
a_p[d] \text{ is } (f \; b_1 \cdots b_n[d] \cdots b_m) \text{ if } p \text{ is } jq, 2 \leq j \leq m + 1, \text{ and } i = j - 1. \\
\text{otherwise, } a_p[d] \text{ is undefined.}
\end{cases}
\]

Substitution for multiple occurrences \( a_{p_1 \cdots p_k}[b] \) is defined to be \( a_{p_i}[b] \) if \( k = 1 \), and \( (a_{p_{k-1}}[b])_{p_k \cdots p_1}[b] \) if \( k > 1 \).

We next define the set \( \Sigma_N(a) \) for each \( N \)-term \( a \) in \( \Lambda^1 \). 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,

\[
\Sigma_N(n) = \{ e \}
\]

\[
\Sigma_N((\text{let} (\langle a \rangle) b)) = \begin{cases} 
3\Sigma_N+1(b) & \text{if } a \in C_N, \\
211\Sigma_N(a) \cup 3\Sigma_N+1(b) & \text{if } a \in \mathcal{R}_N \text{ and } b \in \mathcal{R}_N+1, \\
211\Sigma_N(a) & \text{otherwise}.
\end{cases}
\]

\[
\Sigma_N((\text{set!} n a)) = \begin{cases} 
\{ e \} & \text{if } a \in C_N, \\
3\Sigma_N(a) & \text{otherwise}.
\end{cases}
\]

\[
\Sigma_N((\text{lambda} (\langle n \rangle) a)) = \{ e \}
\]

\[
\Sigma_N((\text{while} a b c)) = 2\Sigma_N(a)
\]

\[
\Sigma_N((\text{if} a b c)) = 2\Sigma_N(a)
\]

\[
\Sigma_N((\text{apply} a b)) = \begin{cases} 
3\Sigma_N(b) & \text{if } a \in C_N, \\
2\Sigma_N(a) \cup 3\Sigma_N(b) & \text{if } a \in \mathcal{R}_N \text{ and } b \in \mathcal{R}_N, \\
2\Sigma_N(a) & \text{otherwise}.
\end{cases}
\]

\[
\Sigma_N((\text{pair} a b)) = \begin{cases} 
3\Sigma_N(b) & \text{if } a \in C_N, \\
2\Sigma_N(a) \cup 3\Sigma_N(b) & \text{if } a \in \mathcal{R}_N \text{ and } b \in \mathcal{R}_N, \\
2\Sigma_N(a) & \text{otherwise}.
\end{cases}
\]

\[
\Sigma_N((f a)) = 2\Sigma_N(b) \text{ where } f \text{ is a term construct not listed above}
\]

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 \text{fun?}, \text{null?}, \text{pair?}, \text{car}, \text{cdr}, \text{and mu}, then
\[
(s \ a) \rightarrow_n (s \ d)
\]

3. If \( a \rightarrow_n d \) and \( b \rightarrow_n e \) and \( s \) is one of \text{pair}, \text{apply}, then
\[
(s \ a \ b) \rightarrow_n (s \ d \ c)
\]

4. If \( a \rightarrow_n d \), \( b \rightarrow_n e \) and \( c \rightarrow_n f \) and \( s \) is \text{if or} \text{while}, then
\[
(s \ a \ b \ c) \rightarrow_n (s \ d \ e \ f)
\]

5. If \( a \rightarrow_n d \), then \( \text{lambda} (\langle a \rangle) a \rightarrow_n (\text{lambda} (\langle a \rangle) d) \)

6. If \( a \rightarrow_n d \), then \( \text{set!} n a \rightarrow_n (\text{set!} n d) \)

7. If \( a \rightarrow_n d \) and \( b \rightarrow_n+1 e \), then \( \text{let} (\langle a \rangle) b \rightarrow_n (\text{let} (\langle d \rangle) e) \)

8. If \( a \in \mathcal{R}_N \), \( s \ a \) is a recognizer term of some kind, and \( a \) is a constructor term of the same kind, then \( (s \ a) \rightarrow_N \text{true} \).

9. If \( a \in \mathcal{R}_N \), \( s \ a \) is a recognizer term of some kind, and \( a \) is a constructor term of a different kind, then \( (s \ a) \rightarrow_N \text{false} \).
Chapter 6
Conclusion

Summary of the thesis
We have studied the paradigm of Constructive Programming based on Sato's Reflective Proof Theory (RPT), and also studied extensions of our framework. RPT is a type-free first-order theory for Constructive Programming. 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 RPT. Our formal system captures essential features of 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 \( \Lambda \), which is at the same time the object language of RPT.
We presented a mechanized proof of the Church-Rosser property of our calculus $\Lambda$. 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 $\Lambda_1$. 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 $\Lambda_1$ as an extension of $\Lambda$ by these statements. $\Lambda_1$ has a close relationship to the $\lambda$-calculus of the explicit substitution by Abadi et al. We studied some conservativeness results on $\Lambda_1$ over the $\lambda$-calculus and the pure $\lambda$-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 $\Lambda_1$. 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


List of Publications by the Author

Major Publications


Oral Presentations


BIBLIOGRAPHY


Other Publications


IPSJ: Information Processing Society, Japan
JSSST: Japan Society for Software Science and Technology
SLACS Workshop: Workshop on Symbolic Logic and Computer Science