Contents

Preface

This book describes a computational logic **PX** (**P**rogram e**X**tractor). **PX** is a constructive logic about computation. The aims of **PX** are to verify programs, extract programs from constructive proofs, and give foundations to type theories. It is well known that programs can be extracted from constructive proofs, but it has not been fully demonstrated how this is actually done, although there are interesting works on the subject. The **PX** project was launched to provide a rigid logical framework and its computer implementation, which enables us to develop methodologies and a theory of program extraction through computer experiments. This book gives a precise description of the formal theory of **PX**, its semantics, the mathematical foundation of program extraction via **PX**, and some methodologies and their theories of program extraction. We also describe its implementation and computer experiments using it.

In the last two decades, metamathematical investigations of constructive logic have made great progress, e.g., Girard's work on higher order constructive logic (Girard 1972), categorical logic, some formal theories developed to formalize Bishop's constructive analysis(Bishop 1967), and the associated metamathematical studies. (See Troelstra 1973, Beeson 1985, Lambek and Scott 1986, and articles in Barwise 1977.) The design of the logical system of **PX** is based on Feferman's theory of functions and classes developed to formalize Bishop's constructive analysis. Martin-Löf's type theory (Martin-Löf 1982) was developed for the same reason. Although these two have much in common, a basic difference exists. Feferman's theory is a *type free* theory and so is **PX**. "Typed or type free" is a controversial issue in programming languages. But in applications of constructive logic to programming, only typed theories have received attention. For example, Constable's group has implemented a Nuprl proof development system taking Martin-Löf's system as its foundation (Constable et al. 1986), and Coquand and Huet designed a theory of construction, which is an extension of Girard-Reynolds type theory (Coquand and Huet 1986). By presenting type free counterparts, we hope to contribute this active area. We wish to show how Feferman's type free theory serves as a logical foundation of programming.

When I was at Tsukuba University, the basic idea of applying Feferman's theory to programming came to me and the predecessors of **PX** were designed (Hayashi 1983). I began the design and implementation of the core of **PX** when I was at Metropolitan College of Technology in Tokyo. Early implementation was done on VAX$^{\mathrm{TM}}$/UNIX$^{\mathrm{TM}}$ at the Computer Center of the University of Tokyo. The project was continued after I moved to Kyoto University, and later Hiroshi Nakano joined the project. We jointly designed the EKL-translator and inference macros, and he implemented them. He also designed and implemented the pretty-printer and corrected a number of errors in my programs. For this book, Nakano has written the section on the EKL-translator, and the part of the proofs of hypotheses by EKL in appendix B.

Many people helped me in the course of the **PX** project. I give thanks to them all here, but I especially thank Prof. Michael J. Beeson, Prof. Solomon Feferman, Dr. Gerard Huet, Prof. Takayasu Ito, Dr. Jussi Ketonen, Zhaohui Luo, Prof. Per Martin-Löf, Dr. Eugenio Moggi, Christine Paulin-Mohring, Dr. Gerard R. Renardel de Lavelette, Prof. Masahiko Sato, Prof. Dana Scott, Dr. Natarajan Shankar, Prof. Satoru Takasu, Dr. Carolyn Talcott, and members of CAP project of ICOT, for their interests in the project and/or helpful discussions, some of which influenced the design of **PX** and this book. I thank Prof. Mike O'Donnell, Prof. Solomon Feferman, James Harland, Dr. Tetsuo Ida, Prof. Takayasu Ito, Dan Pehoushek, Randy Pollak, Dr. Carolyn Talcott, and Prof. Mariko Yasugi for their helpful comments on and criticisms of the manuscript. I thank Japan IBM for its financial support to enable me to present a summary paper at a meeting of IFIP WG2.2, and thank The University of Texas, Austin and Dr. G. Huet for inviting me to lecture on **PX** at The Institute on Logical Foundations of Functional Programming. I also thank the Japanese Ministry of Education (Monbushyo) for supporting this work by a grant. I thank the Department of Computer Science of The University of Edinburgh, the Formel project of INRIA, and the Department of Computer Science of Stanford University for providing me with the facilities to finish the manuscript of this book. Finally, I especially thank Prof. Albert Meyer and Prof. Takayasu Ito for helping and encouraging me to publish this book.

S. Hayashi
Edinburgh
May 1988

1 Introduction

1.1. What is PX?

PX is a constructive logic for computation based on a constructive formal theory T_0 (Feferman 1979). It is a type free logic which formalizes a dialect of pure Lisp. It has rigorous axioms for functional arguments, although it does not allow a reasoning about destructive programs. **PX** has a logic by which one can reason about the termination of programs.

PX has a very large family of data types. The family of data types is much larger than the data types used in the conventional programming languages so that it includes the types in Martin-Löf's sense (except the universes).

PX is a system which realizes the idea of "proofs as programs". When an existence theorem is proved constructively, **PX** extracts a Lisp program that computes a solution. The correctness of the program is certified by a rigorous logical interpretation called **px**-realizability. Program extraction is the main aim of **PX** as the name stands for Program eXtractor.

PX is designed to be compatible with classical logic as far as possible. **PX** provides a way to separate out a "termination proof" in existence theorems. Termination may be proved classically. Programs can be extracted before "termination" is proved. This technique enables us to interpret Hoare logic in **PX** and to prove a relative completeness result.

PX's logic has rigorous mathematical semantics which is compatible with classical mathematics. The semantics guarantees that the solution computed by the extracted program from an existence proof meets the property in the sense of classical mathematics. The semantics guarantees that the extracted function may be run on an ordinary Lisp system insofar as it terminates on the intended domains of inputs. **PX** can translate the termination problem and some other problems to problems in higher order logic and can prove them by a proof checker of classical higher order logic.

As well as basic inference rules, **PX**'s proof checker has an expandable set of inference macros by which highly legible proofs can be written. **PX** enables one to print out proofs written by the macros using $\TeX^{TM}$. So when one proves a theorem with **PX**, not only a Lisp program but also a well formated proof is automatically generated.

1.2. PX as a logic of computation

The logic of **PX** is an ordinary logic except that terms may fail to denote a value. Such a logic is called a logic of partial terms (see 2.3). By aid of this logic, **PX** can reason about the termination of programs. We choose Lisp to describe pure functional algorithms. The Lisp of **PX** is tailored so as to be formalizable without the concepts of environments and situations, so that its logic is still simple enough. The logic of **PX** enables us to reason about functional arguments in Lisp. For example, we can rigorously prove Kleene's second recursion theorem.

Although **PX** is a first order logic, **PX**'s *classes* serve as data types (see 2.4). Classes are defined predicatively as Martin-Löf's types are. The flexibility of a type free system enables us to define Martin-Löf's types and even more data types by two principles, i.e., an induction scheme, called a CIG inductive definition, and a principle of existence of dependent sums. For example, we can define various functional spaces, e.g., spaces of intensional functions, spaces of extensional functions, and spaces of partial functions. We can define many kinds of well-founded structures including Martin-Löf's W (Martin-Löf 1982), and we can formulate a fixed point induction and structural induction by utilizing CIG induction (see 2.4.3 and 4.5). To introduce these into type theories, new types, operators, and inference rules must be added. In our type free approach, all of them can be defined or expressed by the existing language, i.e., they can be introduced without changing the system. (We have to introduce axioms to state the fixed point induction mentioned above, but the axioms are stated in the language of **PX**.)

1.3. PX as a program extractor

The principle of "proofs as programs" maintains that if a constructive proof of $\forall x : D.\exists y.A(x, y)$ is given, then an algorithm of a computable function f is automatically constructed (extracted) from the proof, such that (i) $f(x)$ terminates on every input $x : D$ and (ii) $\forall x : D.A(x, f(y))$ is satisfied. (See Howard 1980 and Bishop 1970.) Since f is constructed from a proof, these properties of f are rigorously certified.

The formula $\forall x : D.A(x, f(y))$ says that if an input x is in the data type D, then the input x and the output $f(x)$ satisfy the relation $A(x, f(x))$. Namely, D specifies a data type on which f is defined and $A(x,y)$ specifies the relation between the input x and the output y. So the formula $\forall x : D.\exists y.A(x, y)$ is a specification of the "extensional" behavior of an algorithm f.

A constructive proof is an amalgamation of an algorithm and its verification. When one develops a program as a proof, even the termination of the program is automatically certified. This is one of the reason why systems like Martin-Löf's type theory appeal to computer scientists. Another reason why such systems appeal is that they can describe many algorithms by a small number of inference rules as formal systems for mathematics. We think that these are advantages, but these can also seen as drawbacks.

In practice, we often need many inductions to extract natural and tractable programs, an we need to separate the termination proof from the part of a proof which represents an algorithm. Functional programs have many different forms of recursion. On the one hand, we need many induction principles to extract such recursions. On the other hand, termination of such recursions can be proved by a small number of induction principles. Namely, there are two different usages of induction. One is to describe recursions and the other is to prove termination of such recursions. These two were confused in the traditional approach of "proofs as programs"

We will show in 3.3 that some natural constructive proofs produce unnatural and slow programs due to this confusion. A typical example is a proof which maintains the existence of a prime number p which is greater than or equal to a natural number n. The proof uses mathematical induction, so the extracted proof uses primitive recursion. The program recursively calls itself $n! - n + 2$ times, even if n is a prime number.

CIG induction provides a way to represent many recursions. **PX** can prove terminations of such recursions before program extraction. Namely, the termination proof can be done at the level of mathematical theory rather than a direct analysis of programs, and so retains the mathematical clarity of "proofs as programs". Furthermore, such a termination proof never disturbs programs to be extracted. Thus, the termination proof of an extracted program can be separated from the other part of the proof. By this method, we can construct a new proof of the prime number example mentioned above by using the old proof as a termination proof, and can extract a simple search program (see 4.3.2.5).

The method of extraction of a recursive program which realizes the formula $\forall x : D.\exists y.A(x, y)$ is as follows. Firstly, we start with an informal recursion scheme which we wish to use in the program. It is not necessary to give details of the program. What must be decided is what recursion *scheme* is used. The instance of the scheme used in the program to be extracted need not be given. We define a subdomain D_1 of D by an instance of CIG induction which represents the scheme. Then we prove $\forall x : D_1.\exists y.A(x, y)$ by induction for D_1. A program f which satisfies $\forall x : D_1.A(x, f(x))$ is extracted from this proof. The proof determines the details of the program. Then f is certified to be total on D_1. The termination of f on D is proved by showing $D \subseteq D_1$. This can be proved by familiar inductions such as those of Martin-Löf's systems. We can prove $\forall x : D.\exists y.A(x, y)$ by this and $\forall x : D_1.\exists y.A(x, y)$. Then the proof of $D \subseteq D_1$ does not disturb f. Hence, the same f is extracted from the proof of $\forall x : D.\exists y.A(x, y)$ (see 3.3).

This is because $D \subseteq D_1$ is a rank 0 formula. Rank 0 formulas are formulas whose proofs never have any computational significance, so it is not necessary to prove them constructively. They may be proved by classical logic or even by another proof checker. **PX** can prove rank 0 formulas by its classical logic (see 2.3.5), or by a proof checker EKL (Ketonen and Weening 1983, 84) of classical higher order logic (see 7.4). Since termination may be proved by classical higher order logic of EKL, all provable recursive functions of higher order arithmetic can be extracted. If we use a proof checker for ZF set theory rather than for higher order logic, then **PX** can extract all provably recursive functions of ZF set theory. There is no theoretical limitation to extensions of the set of extractable programs. **PX** can virtually extract all recursive functions by such extensions. Note that such extensions do not force us to change the logic of **PX**. It is enough to give an interpretation of **PX** in a stronger system. This fact is stated in a rigorous way as a completeness theorem which resembles the relative completeness theorem of Hoare logic (see 4.4). Using a similar argument, we show that Hoare logic can be embedded in **PX** (see 4.6).

1.4. PX as a classical logic

Our philosophy is to use classical logic as much as possible. We restrict ourselves to constructivism only when it is necessary to extract programs from proofs. In practice, it turns out that many lemmas used in proofs (besides termination) are rank 0, so that they may be proved classically. (This was first pointed out by Goad 1980, although he did not consider the termination problem.) **PX** has a simple mechanism by which we can do classical reasoning in the framework of constructive logic. This allows us to extract programs whose termination proofs need classical reasoning. Compatibility with classical reasoning enables us to give a simple classical semantics of formulas, and to relate **PX** to the existing proof checker of classical logic as we did with EKL.

It also allows users to think in the classical way. In constructive type theories, an element f of the type of a formula $\forall x : D.\exists y.A(x, y)$ satisfies $\forall x : D.A(x, f(x))$ in the sense of constructive mathematics. In **PX**, $\forall x : D.A(x, f(x))$ holds in the sense of classical mathematics as well as in the sense of constructive mathematics of **PX**. (See chapters 3 and 4.)

1.5. px/ as a foundation of type theory

Many type theories have been developed in computer science. Their relations are still widely unknown, and there is no one single framework to give semantics to them. We believe that a type free theory will serve as such a framework. The application of type free systems as foundations of type theories is an undeveloped but interesting subject. **PX** itself is not strong enough to capture many type theories, since it is designed as a system of program extraction. Nonetheless, we will show that we can define Martin-Löf's types in **PX** and will point out that **PX** can interpret some module systems. Furthermore, an interpretation of a polymorphic lambda calculus will be given in an impredicative extension of **PX**. (See chapter 5)

1.6. The origin of PX

It seems that quite a few people are puzzled by **PX**'s basic designs. It looks like a logical system designed under the influence of recent studies of type theory in computer science. But it has a rather unconventional flavor. Why untyped theory? Why Lisp? We will explain the origin of **PX** to resolve existing and possible misunderstandings about the motivation of the **PX** project.

Although **PX** looks like a type free counterpart of Martin-Löf's type theory (Martin-Löf 1982) or Nuprl (Constable 1986), it has a different origin. It was born independently of the recent studies of type theory. We were unaware of studies of type theory in computer science in the early stage of the project. The predecessors of **PX** are the studies of a Japanese group (e.g., Goto 1979, 79a; Sato 1979; Takasu and Kawabata 1981; and Takasu and Nakahara 1983). The objective of the group was to give a mathematical foundation to the deductive approach of program synthesis, e.g., Manna and Waldinger 1971, and to build their own systems of program synthesis. The original objective of the **PX** project was to strengthen the studies of the Japanese group both in theory and in its implementation. The main ideas which led us to the **PX** project were replacing the Gödel interpretation by a more natural realizability, to replace HA (Heyting Arithmetic) by much more powerful and natural Feferman's system T_0, and to use Lisp as a target language.

Goto 1979 and Sato 1979 used Gödel interpretation to extract programs from proofs of HA. Sato used reductions of terms to execute extracted primitive recursive functionals. The correctness of the execution was rigorously certified by the correctness of reduction steps. But an experimental reducer written in Lisp was hoplessly slow. Goto's system could compile extracted primitive recursive functionals to natural Lisp programs. But he did not give any correctness proof of the compilation.

One of the authors thought these studies unsatisfactory. Realizability looked much more natural than Gödel interpretation. Goto's approach to generate Lisp codes looked attractive, since extracted programs could run on conventional Lisp systems. But, the extracted programs should be written in the language of logic and the correctness of the extracted programs should be demonstrated by the same logic. It seemed rather easy to formalize properties of a subset of Lisp which is enough for a realizability. Furthermore, there was a very good type free logical system which was quite suitable to formalize the type free language of Lisp. This was Feferman's T_0, which has a similar philosophy to Lisp. The core of both Lisp and T_0 is simple and minimal, but reflect the universe. They are so flexible that their boundaries to the outer world are not clear. They harmonize with the universe, keeping their individuality, similar to Japanese gardens. They are simple and flexible, because they do not distinguish functions and classes (data types) from objects. The key was that they were type free.

So the marriage of T_0 and Lisp was quite natural. The first offsprings of the marriage were **LM** and **LMI** in Hayashi 1983, and an experimental implementation of a subset of **LMI** was built. (**LM** was based on the logic of total terms and **LMI** was based on the logic of partial terms.) A serious drawback of the system was lack of recursive data types. Later, we added recursive data types by means of an extension of Feferman's principle IG (Inductive Generation) to the system. Since the principle looked like a conditional form of Lisp, we called it CIG (Conditional Inductive Generation). Due to this and other changes, the name of the system was changed to **PX**.

Working on some examples, it turned out that CIG provides a technique useful to extract efficient programs. Meanwhile, we realized that it was a way to separate the termination problem from the partial correctness problem, and the metamathematical results in chapter 4 were proved.

During these developments, we realized that the logic of partial terms of **PX** was not satisfactory. Its semantics was based on a complicated translation of logic of partial terms to a logic of total terms with a predicate which denotes the graph of the evaluator of Lisp. And there was a mistake in the treatment of function closures. So we shifted to the present formulation of logic of partial terms and its semantics.

Encouraged by these developments, we tried a larger example presented in appendix b. It was quite difficult to write a sizable proof by basic inference rules of **PX**. We developed inference macros under the influence of the CAP project of ICOT (Furukawa and Yokoi 1984). Inference macros worked well and it turned out that it was easy to print them by $\TeX^{TM}$, and we added the EKL translator to prove many hypotheses which are remained unproved by inference macros. Then the present version of **PX** was born.

1.6. Overview

An overview of this book is as follows. Chapter 2 gives a precise description of the axiom system of **PX**. Chapter 3 gives a realizability that serves as a mathematical foundation for program extraction. These two chapters are rather technical. Readers who are not interested in rigorous mathematical treatments should read chapter 2 briefly and then jump to chapter 4, which gives some examples and methodologies of program extraction and their associated mathematical theories. In chapter 5, we demonstrate that **PX** can serve as a foundation for the semantics of some type theories. Chapter 6 gives a mathematical semantics of **PX**. Chapter 7 describes an implementation of **PX**. Chapter 8 gives a summary of the results of the project and views for further studies. In appendix A we compare our realizability with other interpretations. In appendix B we present experiments with **PX**, in which a tautology checker for propositional logic is extracted from a proof of completeness theorem of propositional logic. Appendix C describes optimization methods that are used in the implementation.

2 Formal System

In this chapter, we introduce the formal theory of **PX**. Basically it is an intuitionistic first order theory axiomatizing properties of a dialect of Lisp.

An important departure of the logic of **PX** from ordinary logic is that in **PX** expressions which may fail to denote a value are admitted. This extension of logic is quite useful when one is attempting to give an axiom system for a computational language, whose programs often fail to terminate. Such a logic was used in classical recursion theory as an informal abbreviation, but was never axiomatized.

In section 2.1, we will introduce a system of expressions, functions, and function definitions. It is a mathematical description of the processes of declaring functions in Lisp. In section 2.2, we will define formulas and terms of **PX**. In sections 2.3, 2.4, and 2.5, we will introduce the axiom system of **PX**.

2.1. DEF system: recursive functions over symbolic expressions

In this section, we will define expressions (terms) and functions of **PX**. The objects described by expressions are *S-expressions*, i.e., objects are generated from atoms by successive applications of dotted pair operations. Namely, we assume that the set of objects *Obj* is inductively generated as follows:

1. Atoms are objects. The set of atoms *Atom* is a disjoint union of the set of natural numbers N and a countably infinite set of *literal atoms*.
2. *Obj* is closed under *pairing* $(* \ . \ *)$ and all objects are generated from atoms by successive applications of pairing.

Before giving the precise definition of expressions and functions, we explain their intended meanings. Roughly, expressions and functions are defined by

$$\begin{aligned}
e ::= &x| \\
&c| \\
&fn(e_1, \ldots, e_n)| \\
&cond(e_1, d_1; \ldots; e_n, d_n)| \\
&\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)| \\
&let\ p_1 = e_1, \ldots, p_n = e_n\ in\ e \\
fn ::= &f| \\
&\lambda(x_1, \ldots, x_n)(e)
\end{aligned}$$

Here $e, e_1, d_1, \ldots$ and fn stand for expressions and a function, respectively, and x ranges over variables, c ranges over constants, and f ranges over function identifiers. These are mathematically refined versions of *M-expressions* (McCarthy 1965) for a certain class of well-formed Lisp programs. **PX** may be thought of as a system based on pure Lisp.

Let us explain the correspondence between our syntax and Lisp programs. (See 6.1 for a detailed explanation.) The application $fn(e_1, \ldots, e_n)$ corresponds to

$$\texttt{(fn e}_1 \ldots \texttt{e}_n\texttt{)}.$$

The conditional expression $cond(e_1, d_1; \ldots; e_n, d_n)$ corresponds to

$$\texttt{(cond (e}_1\texttt{ d}_1\texttt{)}\ldots\texttt{(e}_n\texttt{ d}_n\texttt{))}.$$

The *let*-expression $let\ p_1 = e_1, \ldots, p_n = e_n\ in\ e$ corresponds to

$$\texttt{(let ((p}_1\texttt{ e}_1\texttt{)}\ldots\texttt{(p}_n\texttt{ e}_n\texttt{)) e)}.$$

The abstraction $\lambda(x_1, \ldots, x_n)(e)$ corresponds to

$$\texttt{(lambda (x}_1\ldots\texttt{x}_n\texttt{) e)}.$$

There is no Lisp primitive which precisely corresponds to Λ. But the Λ-expression $\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)$ is almost equivalent to the Common Lisp program

$$\texttt{(let ((x}_1\texttt{ e}_1\texttt{)}\ldots\texttt{(x}_n\texttt{ e}_n\texttt{)) (function fn))}.$$

(See Steele 1984 for Common Lisp.) We will explain Λ in 2.3.1 below in a detail.

The significant difference between **PX** and Lisp is the separation of functions from expressions. In Lisp, a function may be considered to be an expression, but, in **PX**, a function is *not* an expression. An expression of **PX** is a syntactic object whose value is an S-expression. A function of **PX** is a syntactic object whose value is a partial function in the sense of mathematics. Since variables are expressions, we do not have variables for functions. Because of this separation, we must introduce a method for converting functions to expressions to do higher order programming. (Higher order programming is necessary for realizability.) Another possible solution is the introduction of function variables. But it spoils the simplicity of a type free system and does not fit into the semantics of Lisp.

Λ is **PX**'s syntactical operator converting functions to expressions. We use a universal function *app*∗ and an abstraction to convert an expression into a function. (The role of "binding" $x_1 = e_1, \ldots, x_n = e_n$ in $\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)$ will be explained in 2.3.1.) Roughly, if fn is a function, then $\Lambda(fn)$ is an expression such that the function $\lambda(x)(app{*}(\Lambda(fn), x))$ extensionally equals the function fn. (Our notation $\lambda(x)(app{*}(e))$ is $\lambda x.e$ in the usual λ-calculus notation.) So we have two syntactic operations i and j which convert functions and expressions as follows:

$$i(fn) = \Lambda(fn), \quad j(e) = \lambda(x)(app{*}(e, x)),$$

$$\mathit{Func} \underset{j}{\overset{i}{\rightleftarrows}} \mathit{Exp}\,, \qquad [\![j(i(fn))]\!] = [\![fn]\!].$$

Func is the set of functions of **PX**, *Exp* is the set of expressions of **PX**, and $[\![fn]\!]$ is the extension (graph) of the function fn. The expression $app{*}(x, y)$ corresponds to the Lisp program `(funcall x y)` and Kleene's notation $\{x\}(y)$ (see Kleene 1952).

The expressions without recursive definitions can represent any computable function (see proposition 2 in 2.3.2), but programs without recursive definitions are quite slow and consume huge amounts of space. So we introduce recursive definitions of functions as in ordinary Lisp. For a mathematical description of "definitions by recursive equations", we introduce the DEF system, which is a system of Definitions, Expressions, and Functions. (Note that we do not use *label*-notation to define recursive functions.) The definition may look complicated since we insist on a very mathematical description, but it is no more than a subset of Lisp whose grammar, including definitions, is restricted to lexical scoping.

Let B be a countably infinite set of identifiers of basic functions with arities. We assume that at least the following function names belong to B:

$$app,\ app{*},\ list,\ atom,\ fst,\ snd,\ pair,\ equal,\ suc,\ prd.$$

The above ten function names correspond to be the following Lisp functions, respectively:

`apply`, `funcall`, `list`, `atom`, `car`, `cdr`, `cons`, `equal`, `add1`, `sub1`.

Arities of the basic functions are given by a mapping

$$arity_B : B \to N \cup (Pow(N) - (\{\emptyset\} \cup \{\{n\} | n \in N\})),$$

where $Pow(N)$ stands for power set of N. So an arity of a basic function may be a set of natural numbers. The arities of the above basic functions are given by

$$arity_B(app) = 2, \quad arity_B(app{*}) = N - \{0\}, \quad arity_B(list) = N,$$
$$arity_B(atom) = arity_B(fst) = arity_B(snd) = arity_B(suc) = arity_B(prd) = 1,$$
$$arity_B(pair) = arity_B(equal) = 2.$$

Let VV be a countably infinite set of *variables* and C be a countably infinite set of *constants*. In our theory, a variable need not have a value. This is unusual but quite convenient when axiomatizing properties of a programming language. VV is divided into two infinite disjoint sets VV_t and VV_p. The variables of VV_t are called *total variables* and the variables of VV_p are called *partial variables*. We assume total variables (constants) are divided into two disjoint infinite classes, i.e., *class variables* (*class constant*) and *individual variables* (*individual constants*) This division of total variables and constants is not essential, but is convenient for describing the logical system of **PX**. Constants and total variables must have values but partial variables need not have values. Constants are names for fixed *values*. So they have values. (For an abbreviation of a closed expression, which may not have a value, we use 0-argument function instead of a constant.) At least, the following are constants:

$$0,\ t,\ nil,\ V,\ N,\ Atm,\ T,\ quote(\alpha),$$

where α is an arbitrary object, i.e., *quote* is a one-to-one mapping that maps each object to a constant representing it. Among these, V, N, Atm, T are class constants and the others are individual constants. We will often write 'α instead of $quote(\alpha)$. V stands for the class of all objects, N stands for the class of all natural numbers, Atm stands for the class of all atoms, and T stands for the class of all non nil objects.

Constants and partial variables are neither assigned to nor bound by executions of programs. Partial variables may be valueless or may have arbitrary values. Constants are supposed to have fixed values. Partial variables serve as a substitute for metavariables for expressions, since whenever a expression is substituted for a partial variable in a theorem, we still have a theorem. In this sense, the role of partial variables in **PX** is mainly that of enhancing the expressive power of formulas of **PX**. On the other hand, total variables play rather a distinguished role in the **PX** of programming language. They are exclusively used for names of lexical variables. Our intended evaluator can bind lexical variables, so the variable x of $\lambda(x)(e)$ must be a total variable.

Another syntactic category besides B, C, VV is the *function identifier* for user defined functions. We denote it by FI, and we assume that FI is a countably infinite set.

We define the expressions and the functions built from given B, C, VV, and FI. To this end, we define patterns, which are used in *let*-expressions.

Definition 1 (pattern). *Patterns* of *let*-expressions are defined by the following inductive definition. These patterns are a generalization of the usual *let*-patterns of Lisp, so that constants may occur in them.

1. Constants and *total individual* variables are patterns.
2. If $p_1, \ldots, p_n$ are patterns, then $(p_1 \ldots p_{n-1} \;.\; p_n)$ and $(p_1 \ldots p_n)$ are patterns. Note that the last dot of the former pattern is a literal dot and each item of a list is to be separated by spaces.

For example, $(x\ y)$, x, $(x\ y\ .\ x)$, $(x\ 0)$, $(x\ (y)\ .\ nil)$ are patterns. The free variables (or variables for short) of a pattern are the variables appearing in it. We denote the set of variables of a pattern p by $FV(p)$. For example, $FV(((x\ y)\ x\ 0))$ is $\{x,\ y\}$. Patterns are useful to avoid messy destructors *fst* and *snd* as patterns of some Lisp systems are used. We allow constants to appear in patterns for the compatibility with the usage of patterns in ∇-quantifier (see 2.2).

Now we can define expressions and functions.

Definition 2. Let A be a *finite* set of function names with an arity function $arity_A : A \to N$. We assume A is disjoint to B. We give an inductive definition of the *expressions over* A, $E(A)$ for short, and the *functions over* A, $F(A)$ for short. Simultaneously we define their free variables and arities of functions. The set of free variables of α is denoted by $FV(\alpha)$. The arity is defined as a function

$$arity : F(A) \to N \amalg (Pow(N) - (\{\emptyset\} \cup \{\{n\} | n \in N\}),$$

1. $C,\ VV \subseteq E(A)$. If $\alpha \in C$, then $FV(\alpha)$ is $\{\ \}$, and, otherwise, it is $\{\alpha\}$.
2. Assume $\alpha \in A \cup B$. Then $\alpha \in F(A)$, $FV(\alpha) = \emptyset$, and its arity is defined as $arity_A(\alpha)$ or $arity_B(\alpha)$.
3. Assume $f \in F(A)$ such that $arity(f) = n$ or $n \in arity(f)$. If $e_1, \ldots, e_n \in E(A)$, then
$$f(e_1, \ldots, e_n)$$
belongs to $E(A)$ and its free variables are $FV(f) \cup FV(e_1) \cup \ldots \cup FV(e_n)$. This is called an *application*.
4. Assume $e_1, \ldots, e_n, d_1, \ldots, d_n \in E(A)$. Then
$$cond(e_1, d_1; \ldots; e_n, d_n)$$
belongs to $E(A)$. This is called a *conditional expression*. Its free variables are $FV(e_1) \cup FV(d_1) \cup \ldots \cup FV(e_n) \cup FV(d_n)$. We will use $case(e, e_1, \ldots, e_n)$ as an abbreviation of
$$cond(equal(e, 1), e_1; \ldots; equal(e, n), e_n).$$
5. Assume $f \in F(A)$ and all of free variables of f are *total variables*. Let $v_1, \ldots, v_n$ be a sequence of total variables without repetitions such that all free variables of f appear among them. Then
$$\Lambda(v_1 = e_1, \ldots, v_n = e_n)(f)$$

belongs to $E(A)$. This is called a Λ-*expression* and its free variables are $FV(e_1) \cup \ldots \cup FV(e_n)$. $\Lambda(f)$ is an abbreviation of $\Lambda(v_1 = v_1, \ldots, v_n = v_n)(f)$, where $FV(f) = \{v_1, \ldots, v_n\}$. The variables $v_1, \ldots, v_n$ are called the bound variables of the Λ-expression, and the part of $v_1 = e_1, \ldots, v_n = e_n$ is called the binding of the Λ-expression. The function f is called the body of the Λ-expression.

6. Assume $e \in E(A)$ and $v_1, \ldots, v_n$ are *total individual variables* without repetitions. Then
$$\lambda(v_1, \ldots, v_n)(e)$$
belongs to $F(A)$. This is called a λ-*function* and its free variables are $FV(e) - \{v_1, \ldots, v_n\}$. (It is not an "expression" but a "function" according to our terminology. So we call it a λ-function instead of λ-expression.) Sometimes we will write $\lambda(v_1, \ldots, v_n).e$ for simplicity. The variables $v_1, \ldots, v_n$ are called the bound variables of the λ-function. The expression e is called the body of the λ-function.
7. Assume $e_1, \ldots, e_n, e$ belong to $E(A)$ and $p_1, \ldots, p_n$ are patterns. Then
$$let\ p_1 = e_1, \ldots, p_n = e_n\ in\ e$$
belongs to $E(A)$. This is called a *let-expression* and its free variables are $FV(e) - (FV(p_1) \cup \ldots \cup FV(p_n)) \cup FV(e_1) \cup \ldots \cup FV(e_n)$. The variables of $p_1, \ldots, p_n$ are called the bound variables of the *let*-expression. The expression e is called the body of the *let*-expression.

The syntactic constructors Λ, λ, and *let* are *quantifiers*. Λ-expressions and *let*-expressions are *quantified expressions*. λ-functions are *quantified functions*. An occurrence of a variable in an expression or function is bound in it, when it is a bound variable of a subquantified expression (or a subquantified function) and in the body of the subquantified expression (or the subquantified function). An occurrence of a variable is free, unless it is bound. A bound occurrence of a variable is bound by the same bound variable of the quantifier that is the smallest quantified expression (or quantified expression) in which the occurrence is bound.

Substitution of expressions for variables of an expression or a function are defined as usual. The expression obtained by substituting $e_1, \ldots, e_n$ for variables $x_1, \ldots, x_n$ of e is denoted by $e[e_1/x_1, \ldots, e_n/x_n]$. Not that we do not substitute a function for a variable, since all variables range over S-expressions rather than functions. Since a conflict of free variables and bound variables may occur, renaming of bound variables is necessary. The equivalence of expressions or functions by α-convertibility, which involves semantics of the DEF system, will be discussed in 2.3.1.

We give a formal definition of "definition of functions".

Definition 3 (definition of functions). Let $f_1, \ldots, f_n$ be function identifiers without repetitions. A *definition* of $f_1, \ldots, f_n$ over A is a set of equations $\{f_1(\vec{v})_1 = e_1, \ldots, f_n(\vec{v})_n = e_n\}$ such that $\vec{v}_i$ is a sequence of variables without repetitions, $e_i \in E(A)$, and $FV(e_i) \subseteq \vec{v}_i$. We assume the arity of each function f_i matches to the number of arguments.

Now we give the definition of DEF system.

Definition 4 (finite DEF system). A triple D, E, F is a *finite DEF system* iff it is defined by one of the following clauses:

1. D, E, F are $\emptyset, E(\emptyset), F(\emptyset)$, respectively.
2. Assume $\langle D, E, F\rangle$ is a finite DEF system. Let $f_1, \ldots, f_n$ be new function identifiers not belonging to F. Let A be the union of the set of function identifiers of F and $\{f_1, \ldots, f_n\}$. Then for each definition d over A, $\langle D \cup \{d\}, E(A), F(A)\rangle$ is a finite DEF system. We denote this new finite DEF system by $\langle D, E, F\rangle \oplus d$.

We define the DEF system as a limit of finite DEF systems.

Definition 5 (DEF system). Let $DEF_0, DEF_1, \ldots$ be a infinite sequence of finite DEF systems such that each DEF_{i+1} is $DEF_i \oplus d_{i+1}$ for a definition d_{i+1} of new functions and DEF_0 is $\langle \emptyset, E(\emptyset), F(\emptyset)\rangle$. Let DEF_i be D_i, E_i, F_i. Then $\langle \bigcup_i D_i, \bigcup_i E_i, \bigcup_i F_i\rangle$ is called a *DEF system*. A DEF system is called *regular* if all possible definitions appear in the sequence $d_1, d_2, \ldots$ modulo renaming of function names.

We can write all recursive programs in a regular DEF system. In 2.3.1, we will show that even in the least finite DEF system $\langle \emptyset, E(\emptyset), F(\emptyset)\rangle$ we can define *all* recursive functions by the aid of Kleene's second recursion theorem. It is easy to see there is a regular DEF system. A regular DEF system is practically unique in the sense that any functions of any DEF system are interpretable by any other regular DEF system. So it is not essential which DEF system is chosen.

Remark 1. A finite DEF system corresponds to a finite collection of definitions of functions in **PX** proof checker (see 7.2.6). But we consider that **PX** is based on a regular DEF system rather than on a finite DEF system, in order to state the soundness theorem for our realizability (theorem 1 of 3.1) naturally. If **PX** is based on a finite DEF system and a theorem is proved in it, then the realizer of the theorem may belong to a larger finite DEF system rather than the DEF system that **PX** is based on, since the extraction algorithm may define new functions to construct the realizer. But a realizer extracted from a theorem should belong to the same system in which the theorem is proved. Thus we use a regular DEF system instead of a finite DEF system.

2.2. The language of PX

PX is a system in which we can write recursive programs and prove their properties. Such a system must have two parts, a computational part and a logical part. The computational part is a programming language and the logical part is a logical system in which we can state and prove the properties of programs and data types. In Hoare logic, the computational part is called the object language and the logical part is called the assertion language. Any system of program verification must have these two parts. A single framework may unify these two parts (see, e.g., Sato 1985), but computing and logical reasoning about them have rather different natures and different aims. So we separate them.

In this section we will present the languages of both the computational and the logical parts of **PX**. The axiom system will be explained in the following sections 2.3-2.5. The precise semantics will be given in chapter 6, although we will give informal explanations in this and the following sections. **PX** is a two-sorted first order constructive formal system. Fix a regular DEF system $\langle D, E, F\rangle$, and we define the **PX** system on it. The terms (expressions) of **PX** are the expressions of E and the functions are the functions of F. Since a regular DEF system is not unique, the formal system of **PX** is not unique, either. This reflects the fact that we can declare functions in the implementation of **PX**, so that definitions of terms and axioms of **PX** depend on declarations. To define formulas of **PX** we must define some auxiliary concepts.

Definition 1 (expansion). *Expansions of patterns* are defined inductively by the following clauses. The expansion of a pattern p will be denoted by $exp(p)$.

1. The expansions of constants and total variables are themselves.
2. Compounded patterns are expanded as follows:

$$\begin{aligned} exp((p_1 \ . \ p_2)) &= pair(exp(p_1), exp(p_2)), \\ exp((p_1 \dots p_{n-1} \ . \ p_n)) &= pair(exp(p_1), exp(p_2 \dots p_{n-1} \ . \ p_n)), \\ exp((p_1 \dots p_n)) &= list(exp(p_1), \dots, exp(p_n)). \end{aligned}$$

Note that an expansion of a pattern p is an expression constructed from total constants and total variables by successive applications of the function identifiers *pair*, *list*. The *free variables of a pattern p* are just the free variables of $exp(p)$.

Definition 2 (tuple). If $e_1, \dots, e_n$ are expressions, then $[e_1, \dots, e_n]$ is a *tuple*. We do not allow nested tuples, e.g., $[e_1, [e_2]]$ is not a tuple in our sense. Tuple is a meta notation (or an abbreviation) so it is not a part of formal syntax. We identify $[e]$ with e itself. On the other hand, a tuple $[e_1, \dots, e_n]$ whose length is not 1 stands for $list(e_1, \dots, e_n)$. Especially, $[\,]$ is *nil*. A tuple must be used only in the context of the form $[e_1, \dots, e_n] : e$ or as a pattern. The free variables of

$[e_1, \ldots, e_n]$, symbolically $FV([e_1, \ldots, e_n])$, are $FV(e_1) \cup \ldots \cup FV(e_n)$. If $v_1, \ldots, v_n$ are mutually distinct total variables, then the tuple $[v_1, \ldots, v_n]$ is called a *tuple variable*. This may be used as a bound variable of a quantifier. We will often denote a tuple variable by the vector notation $\vec{v}$.

Definition 3 (formula). In the following $e, d, e_i, d_i \ldots$ are metavariables for expressions.

1. The atomic formulas of **PX** are

$$E(e),\ \mathit{Class}(e),\ e_1 = e_2,\ [e_1, \ldots, e_n] : e_{n+1},\ \bot,\ \top.$$

   We write $e_1 : e_2$ for $[e_1] : e_2$.
2. If $A_1, \ldots, A_n$ are formulas then so are

$$A_1 \vee \ldots \vee A_n, \quad A_1 \wedge \ldots \wedge A_n.$$

3. If A and B are formulas, then so is

$$A \supset B.$$

   $A \supset\subset B$ is an abbreviation of the formula $A \supset B \wedge B \supset A$.
4. If A is a formula, then so are

$$\neg A, \quad \Diamond A.$$

5. If A is a formula, $e_1, \ldots, e_n$ are expressions, and $\vec{v}_1, \ldots, \vec{v}_n$ are tuple variables such that $FV(\vec{v}_1), \ldots, FV(\vec{v}_n)$ are mutually disjoint, then so are

$$\forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A, \quad \exists \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A.$$

6. If $e_1, \ldots, e_n$ are expressions and $A_1, \ldots, A_n$ are formulas, then so is

$$e_1 \to A_1; \ldots; e_n \to A_n$$

   A formula of this form will be called a *conditional formula* and $e_i \to A_i$ is called a clause with condition e_i and body A_i.
7. If A is a formula, $p_1, \ldots, p_n$ are patterns, and $e_1, \ldots, e_n$ are expressions such that $FV(p_1), \ldots, FV(p_n)$ are mutually disjoint, then so is

$$\nabla p_1 = e_1, \ldots, p_n = e_n.A.$$

   ∇ is called ∇-quantifier or *let*-quantifier.

In short, formulas are defined by the grammar:

$$\begin{aligned}F ::= {} & E(e)|Class(e)|[e_1,\ldots,e_n] : e|e_1 = e_2|\top|\bot| \\ & F_1 \wedge \ldots \wedge F_n|F_1 \vee \ldots \vee F_n|F_1 \supset F_2|e_1 \rightarrow F_1;\ldots;e_n \rightarrow F_n|\neg F|\Diamond F| \\ & \forall\vec{x}_1 : e_1,\ldots,\vec{x}_n : e_n.F|\exists\vec{x}_1 : e_1,\ldots,\vec{x}_n : e_n.F|\nabla p_1 = e_1,\ldots,x_n = e_n.F,\end{aligned}$$

where $F, F_1, \ldots$ are formulas and $e, e_1, \ldots$ are expressions. We will casually use parentheses to group formulas.

Now we explain the meaning of the formulas informally. $E(e)$ means e has a value or the execution of e under the current environment terminates. $Class(e)$ means that e has a value which is a class. Classes are descriptions of particular sets of objects. Although precise definition of classes will be given later, readers may here assume that any reasonable data type, such as the natural numbers, lists, function spaces, etc., are classes. The equality $e_1 = e_2$ is Kleene's equality, i.e., if e_1 has a value then e_2 has the same value and vice versa. $[e_1,\ldots,e_n] : e$ means that $e_1,\ldots,e_n,e$ have values, say $v_1,\ldots,v_n,v$, respectively, and v is a description of a class to which the tuple $[v_1 \ldots v_n]$ belongs. Namely, if $n = 1$, then it means v_1 belongs to v, and if $n \neq 1$, then it means the list $(v_1 \ldots v_n)$ belongs to v. Note that $[e_1] : e$ does *not* mean the singleton list (v_1) belongs to v. $\top$ and $\bot$ mean true and false, respectively.

The logical connectives except $\rightarrow$, $\Diamond$ are as usual. The modal operator $\Diamond$ is just double negation. $e_1 \rightarrow A_1;\ldots,e_n \rightarrow A_n$ means there is m such that $e_1 = \ldots = e_{m-1} = nil$ and e_m has a non nil value and A_m holds. Note that if all of $e_1,\ldots,e_n$ have the value nil then the formula is false. Since this semantics resembles McCarthy's conditional form, we call this formula a conditional formula. As in the case of conditional form we can define "serial or" as $e_1 \rightarrow \top;\ldots;e_n \rightarrow \top$. We will abbreviate it as $Sor(e_1,\ldots,e_n)$. The case formula $Case(a, A_1,\ldots,A_n)$ is an abbreviation of $equal(a,1) \rightarrow A_1;\ldots;equal(a,n) \rightarrow A_n$.

The universal quantifier $\forall\vec{v}_1 : e_1,\ldots,\vec{v}_n : e_n A$ means that if $\vec{v}_1 : e_1,\ldots,\vec{v}_n : e_n$ hold, then A holds. For example, $\forall[x,y] : C.R(x,y)$ means that $R(x,y)$ holds for all x, y such that $[x,y] : C$. Since we identify a tuple variable $[x]$ with x, we write $\forall x : e.A$ instead of $\forall[x] : e.A$. We abbreviate $\forall\ldots,x : V,\ldots$ as $\forall\ldots,x,\ldots$. So $\forall[x] : V,[y] : V.A$ will be abbreviated as $\forall x,y.A$. Note that $\forall x_1 : e_1, x_2 : e_2.A$ is not equivalent to $\forall x_1 : e_1.\forall x_2 : e_2.A$, since, in the former formula, the free occurrences of the variable x_1 in e_2 are not bound. The semantics of the existential quantifier is similar to that of the universal quantifier.

The ∇-quantifier resembles *let* of Lisp. $\nabla p = e.A$ means that e has a value which matches the pattern p and that under the matching A holds. The scope of the ∇-quantifier is the same as that of the universal quantifier. We will sometimes use a tuple notation for expressing a pattern. Then the tuple is a macro which expands by the rule in Definition 2 above. Namely, $\nabla[p] = e.A$ means $\nabla p = e.A$, and $\nabla[p_1, \ldots, p_n] = e.A$ means $\nabla(p_1, \ldots, p_n) = e.A$, if n is greater than 1. The ∇-quantifier is quite useful in specifications of programs in symbolic manipulations like programs on logic. In such specifications, we often say as "let A be a formula of the form $B \wedge C \cdots$". This can be stated as $\nabla(B \text{ '}\wedge C) = A.(\cdots)$. It is also useful for a formal definition of realizability. Its inference rules are used to extract the program construct *let*.

Note that we can define ∇ by means of the universal quantifier and also by means of the existential quantifier. For example, $\nabla x = e.A$ is defined by $E(e) \wedge \forall x.(x = e \supset A)$ and also by $\exists x.(x = e \wedge A)$. But our realizability of $\nabla x = e.A$ is different from the realizability of these two formulas. This is one of the reasons why ∇ is a primitive logical sign. We can define $\rightarrow$ and the predicate E by means of the other logical signs. But they are primitive by the same reason.

We will assign a natural number $rank(A)$ called *rank of* A for each formula A in 3.2. Roughly, a rank of a formula is the length of lists realizing the formula (see chapter 3 for realizability). The formulas whose rank is 0, which we call rank 0 formulas, play an important role in **PX**. The rank 0 formula generalizes the Harrop formula, the almost negative formula of Troelstra 1973, and the normal formula of Nepeĭvoda 1978. (Troelstra allows the use of $\exists$ in front of a recursive predicate in almost negative formulas. Such a quantifier may be replaced by a ∇-quantifier.) To assert a rank 0 formula, it is enough to show it holds in the sense of classical mathematics. Here we define rank 0 formulas by the following grammar.

Definition 4 (rank 0 formula). Let F range over all formulas and let $G, G_1, \ldots$ range over rank 0 formulas. Then *rank 0 formulas* are generated by the grammar:

$$\begin{aligned} G ::= &E(e) | Class(e) | [e_1, \ldots, e_n] : e | e_1 = e_2 | \top | \bot | \\ &G_1 \wedge \ldots \wedge G_n | F \supset G | e_1 \rightarrow G_1; \ldots; e_n \rightarrow G_n | \neg F | \Diamond F | \\ &\forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.G | \nabla p_1 = e_1, \ldots, x_n = e_n.G. \end{aligned}$$

In 4.6, we will show how to simulate Hoare logic in a fragment of **PX**. There, rank 0 formulas play the role of formulas of assertion language. As a generalized counterpart of the consequence rule of Hoare logic, we will introduce the rule of (*replacement*) in 2.3.3, which enables us to replace a rank 0 subformula in a context by another subformula. For the formulation of the inference rule, we define the concept of *contexts* and their *environments*.

Definition 5 (context). An $*$-formula is a formula which allows $*$ as an atomic formula. A *context* is an $*$-formula which has exactly one occurrence of $*$. We write a context as $A[*]$. If $A[*]$ is a context and F is a formula, then $A[F]$ is a formula obtained by replacing $*$ by F.

Note that the symbol $*$ may occurs only once in a context $A[*]$. So $* \wedge *$ is an $*$-formula but not a context. When $*$ occurs in a positive (negative) part of $A[*]$, we write $A[*]_+$ ($A[*]_-$). For example, the following are contexts:

$$A[*]_+ = \forall x : C(x = y \supset *),\ A[*]_- = \forall x : C(* \supset x = x).$$

Note that we do *not* rename bound variables of $A[F]$. So $A[x = x]$ in the above example is $\forall x : C(x = y \supset x = x)$.

Definition 6 (environment). A context called the *environment* of $*$ in $A[*]$, $Env_A[*]$ for short, is defined by the clauses:

1. $Env_*[*]$ is $*$.
2. If $A[*]$ is a conjunctive context, then just one conjunct is a context, i.e., $A[*]$ has the form $A_1 \wedge \ldots \wedge A_i[*] \wedge \ldots \wedge A_n$, where the conjuncts $A_1, \ldots, A_n$ are ordinary formulas except A_i. Then $Env_A[*]$ is $Env_{A_i}[*]$. The environment of a context whose outermost logical symbol is $\vee, \supset, \neg, \Diamond$ or $\rightarrow$ is defined similarly.
3. If $A[*]$ is a context $\forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.B[*]$ or $\exists \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.B[*]$, then its environment is $\forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.Env_B[*]$. If $A[*]$ is $\nabla p_1 = e_1, \ldots, p_n = e_n.B[*]$, then its environment is $\nabla p_1 = e_1, \ldots, p_n = e_n.Env_B[*]$.

2.3. Basic axiom system: the logic of partial terms

In this section, we introduce a basic axiom system describing the properties of programs (expressions and functions) and logical symbols. The axiom system concerning data types will be presented in the next section. The axiomatic system in this section is a logic of partial terms (LPT). In the usual logic, every expression (term) is assumed to have a value. But in computational languages, expressions (programs) often fail to have a value due to loop or error. So usual logic is not too useful in axiomatizing properties of such a language. There are many attempts to axiomatize computational language: Hoare logic, dynamic logic, etc. Many of these attempts concern the axiomatization of imperative languages. LPT aims at axiomatizing a pure functional language. One advantage of our approach is that it is quite similar to ordinary logic; with only one additional logical primitive, asserting the termination of a program, we can formulate practically almost all properties of programs by means of the usual symbolism.

Similar and virtually the same formulations of LPT have been presented by various authors independently and with different motivations (e.g., Fourman 1977; Scott 1979; Beeson 1981, 1985, 1986; Hayashi 1983, 1986; and Plotkin 1985).

(Fourman and Scott's logic is not a logic of partial terms but a logic of *partial existence*. See 2.6 for a discussion of the difference.) Their homogeneity would be an evidence that LPT is *the* logic of partial terms. (See Moggi 1988 for an extensive study on LPT.)

2.3.1. Λ-expression and α-convertibility

Before going into the axiom system, we will examine the role of Λ-expression and the definition of α-convertibility.

As we pointed out in 2.1, $\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)$ is almost equivalent to

$$\texttt{(let ((x}_1\texttt{ e}_1\texttt{)}\ldots\texttt{(x}_n\texttt{ e}_n\texttt{)) (function fn))}.$$

It is evaluated to an S-expression as follows:

1. When fn is a function identifier, $\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)$ is evaluated to a code of the function identifier fn.
2. When fn is $\lambda(y_1, \ldots, y_m)(e)$, $\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)$ is evaluated to a code of the following function

$$\lambda\ (y_1, \ldots, y_m)(let\ x_1 = quote(v_1), \ldots, x_n = quote(v_n)\ in\ e)$$

   where v_i is the value of e_i. (See 6.1 for code.) In the second case, when one of $e_1, \ldots, e_n$ does not have a value, its value is undefined. Then the value of the expression is undefined.

It looks enough that introducing a syntactic constructor $\Lambda(fn)$ corresponding to (`function fn`) and defining $\Lambda(x_1 = e_n, \ldots, x_n = e_n)(fn)$ by $let\ x_1 = e_n, \ldots, x_n = e_n\ in\ \Lambda(fn)$. But this contradicts to equality rule and our intended semantics. Our intended semantics of $\Lambda(fn)$ is the value of a functional argument (`function fn`). So the value of (`function fn`) consists of a code of fn and its environment. Let us show how this contradicts to simple $\Lambda(fn)$. By equality axiom,

$$\Lambda(\lambda(x)(y)) = \Lambda(\lambda(x)(y))$$

holds and y is a free variable of $\Lambda(\lambda(x)(y))$. So, if we substitute $fst(pair(0,0))$ and 0 for y of the left hand side and y of the right hand side, respectively, then we will have

$$\Lambda(\lambda(x)(fst(pair(0,0))) = \Lambda(\lambda(x)(0)),$$

for $fst(pair(0,0)) = 0$ holds. But this does not holds, for the codes of the two expressions $\lambda(x)(fst(pair(0,0)))$ and $\lambda(x)(0)$ are different S-expressions

```
(lambda (x) (fst (pair 0 0)))   and   (lambda (x) 0).
```

By introducing $\Lambda(x_1 = e_n, \ldots, x_n = e_n)(fn)$, we can avoid this problem. Since all of free variables of fn of $\Lambda(x_1 = e_n, \ldots, x_n = e_n)(fn)$ must appear among $x_1, \ldots, x_n$, $\Lambda(\lambda(x)(y))$ must be $\Lambda(y = y)(\lambda(x)(y))$. If we substitute $\mathit{fst}(\mathit{pair}(0,0)))$ and 0 for y of $\Lambda(\lambda(x)(y))$, then it becomes

$$\Lambda(y = \mathit{fst}(\mathit{pair}(0,0)))(\lambda(x)(y))$$

and

$$\Lambda(y = 0)(\lambda(x)(y)).$$

Since $\mathit{fst}(\mathit{pair}(0,0))$ is evaluated to 0 when

$$\Lambda(y = \mathit{fst}(\mathit{pair}(0,0)))(\lambda(x)(y))$$

is computed, the values of the both Λ-expressions become the S-expression

```
(lambda (x) (let ((y 0)) y)).
```

The point is that all of free variables of the body fn of a Λ-expression are bound by the Λ-expression so that any substitution cannot be made for them.

A similar problem appears with respect to α-convertibility of Λ-expressions. For example, the two Λ-expressions $\Lambda()(\lambda(a)(a))$, and $\Lambda()(\lambda(b)(b))$ are *not* α-convertible, although they are expected to be so, since their values are different S-expressions `(lambda (a) a)` and `(lambda (b) b)`, respectively. So the bodies of two α-convertible Λ-expressions must be *identical*. Thus, α-convertibility of expressions and functions are defined as follows:

Definition 1 (α-convertibility)

1. Two constants (variables) are α-convertible iff they are identical.
2. Two expressions constructed by application or *cond* are α-convertible iff the corresponding functions and subexpressions are α-convertible.
3. *let* $p_1 = e_1, \ldots, p_n = e_n$ *in* e and *let* $q_1 = d_1, \ldots p_n = d_n$ *in* d are α-convertible iff there is a substitution σ such that
   (i) $\sigma(a) = \sigma(b)$ iff $a = b$, i.e., σ is bijective,
   (ii) e_i and d_i are α-convertible and $p_i\sigma = q_i$, for each $i = 1, \ldots, n$,
   (iii) $e\sigma$ and d are α-convertible.
4. α-convertibility of λ-functions is similarly defined.
5. $\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)$ and $\Lambda(y_1 = d_1, \ldots, y_n = d_n)(gn)$ are α-convertible, iff $x_1, \ldots, x_n, fn$ and $y_1, \ldots, y_n, gn$ are *literally identical*, respectively, and e_i and d_i are α-convertible for each $i = 1, \ldots, n$.

Since all the free variables of the body of a Λ-expression are bound, no renaming of bound variables of the function fn and the bound variables $x_1, \ldots, x_n$ of $\Lambda(x_1 =$

$e_1, \ldots, x_n = e_n)(fn)$ is done even if a substitution is made. So the above definition of α-convertibility is consistent with renaming of variables by substitution.

The difficulty we examined above is not so relevant to actual Lisp programming, for two functions closures are seldom compared by *equal*. Whenever we do not care about equivalence of two function closures as S-expressions, we can neglect this problem. But for consistent axiomatization, we have to take care of such a difficulty as far as it may occur in principle, even if it may not be relevant to actual programming activity.

The α-convertibility of formulas is defined as usual from the α-convertibility of expressions. Using the inference rule of (*alpha*) below, we may rename any bound variables of a formula insofar as the result of renaming is α-convertible to the original one. So, when we make a substitution, we may assume that renamings of bound variables are automatically done. The formula obtained by substituting $e_1, \ldots, e_n$ for variables $x_1, \ldots, x_n$ of A is denoted by $A[e_1/x_1, \ldots, e_n/x_n]$.

2.3.2. Axioms for programs and primitive predicates

The following give the basic axioms and inference rules about programs and primitive predicates. Although they are stated for a fixed algorithmic language, they may be regarded as a general formalization of an axiomatic recursion theory. The key of our axiom system is the treatment of the Λ-expression. Our Λ-expression is essentially the same as the Λ-notation of Kleene 1952, and it is another description of the S_n^m function, which is the key to the axiomatization of recursion theory. (See Friedman 1971 for axiomatic recursion theory.)

The axioms and rules are presented in natural deductive form. Our notation for natural deduction is slightly different from the usual one. We use sequents to present a theorem in a natural deduction. For example, the fact "F is derivable from the assumptions $A_1, \ldots, A_n$" is expressed in such a way that $\{A_1, \ldots, A_n\} \Rightarrow F$ is derivable. The right hand side of a sequent is a single formula and the left hand side is a *set* of a formula. So we do not need any structural inference such as permutations of assumptions. The formulas of the left hand side of a sequent are called the assumptions and the right hand side is called the conclusion. Each inference rule is displayed as

$$\frac{S_1, \ldots, S_n}{S},$$

where $S_1, \ldots, S_n, S$ are sequents. $S_1, \ldots, S_n$ are called the *upper sequents* of the rule and S is called the *lower sequent*. We will sometimes call the upper sequents the premises. Sometimes the rules will be displayed as $S_1, \ldots, S_n/S$ to save space. Note that some inference rules does not have upper sequents, i.e., $S_1, \ldots, S_n$ are absent. Then the lower sequent of the rule is called an *initial sequent* and the rule is displayed by the initial sequent S instead of

$$\overline{S}$$

Axioms are initial sequents without assumptions and they are displayed as a single formula. All of the inference rules in this subsection 2.3.2 are axioms except $(=4)$. $(assume)$ and $(\top)$ of 2.3.3 are initial sequents.

Now we display the axioms and rules. In the following, $a, b, a_1, \ldots, a_n$ stand for *total* variables, $e, e_1, \ldots, e_n, d_1, \ldots, d_n$ stand for expressions, $p_1, \ldots, p_n$ stand for patterns, and fn stands for a function.

$(E1)$ $$E(e) \quad (e \text{ is a constant or a total variable})$$

$(E2)$ $$Class(e) \supset E(e)$$

$(E3)$ $$[e_1, \ldots, e_{n-1}] : e_n \supset E(e_i) \quad \text{for } i = 1, \ldots, n$$

$(E4)$ $$E(fn(e_1, \ldots, e_n)) \supset E(e_1) \wedge \ldots \wedge E(e_n)$$

$(E5)$ $$E(e_1) \wedge \ldots \wedge E(e_n) \supset E(\Lambda(v_1 = e_1, \ldots, v_n = e_n)(fn))$$

$(Class1)$ $$Class(e) \quad (e \text{ is a class constant of a class variable})$$

$(Class2)$ $$[e_1, \ldots, e_n] : e \supset Class(e)$$

$(=1)$ $$e = e$$

$(=2)$ $$e_1 = e_2 \supset e_2 = e_1$$

$(=3)$ $$(E(e_1) \vee E(e_2) \supset e_1 = e_2) \supset e_1 = e_2$$

$(=4)$ $$\frac{\Gamma \Rightarrow P[e_1/a] \qquad \Pi \Rightarrow e_1 = e_2}{\Gamma \cup \Pi \Rightarrow P[e_2/a]}$$

(= 5) $$a = b \vee \neg a = b$$

(*app*) Let σ be the substitution $[a_1/v_1, \ldots, a_n/v_n]$. Then the following are axioms:

$$\mathit{app}(\Lambda(v_1 = a_1, \ldots, v_n = a_n)(fn), \mathit{list}(e_1, \ldots, e_n)) = (fn\sigma)(e_1, \ldots, e_n)$$
$$\mathit{app}{*}(\Lambda(v_1 = a_1, \ldots, v_n = a_n)(fn), e_1, \ldots, e_n) = (fn\sigma)(e_1, \ldots, e_n)$$

(*beta*) $E(e_1) \wedge \ldots \wedge E(e_n) \supset (\lambda(v_1, \ldots, v_n)(e))(e_1, \ldots, e_n) = e[e_1/v_1, \ldots, e_n/v_n]$

(*cond*)
$$\mathit{cond}(e_1, d_1; \ldots; e_n, d_n) = a \;\supset\subset\; e_1 \to a = d_1; \ldots; e_n \to a = d_n; t \to a = \mathit{nil}$$

(*let*) For each a not belonging to $FV(p_1) \cup \ldots \cup FV(p_n)$, the following is an axiom:

$$(\mathit{let}\ p_1 = e_1, \ldots, p_n = e_n\ \mathit{in}\ e) = a \;\supset\subset\; \nabla p_1 = e_1, \ldots, p_n = e_n . e = a$$

By the axioms (= 1), (= 3), we see our equality is Kleene's equality instead of the equality defined as both sides having the same value. For example, if e_1 and e_2 are undefined, then $e_1 = e_2$ holds by the axiom (= 3). (= 5) says the equality is decidable for total variables, i.e., values.

The axiom (*beta*) is call-by-value β-conversion. The axiom ($E4$) ensures that our computation rule for application is call-by-value.

The value of an expression e can be characterized by determining the extension of the set $\{a | e = a\}$. This is the way by which we characterized the values of *let* and *cond* expressions above. This gives a way of proving equations. If the extensions of $\{a | e_1 = a\}$ and $\{a | e_2 = a\}$ are extensionally equal, then $e_1 = e_2$ holds. Indeed, the following proposition holds.

Proposition 1. *In* **PX** *we can prove that $e_1 = e_2$ holds iff*

$$\forall\, a.(e_1 = a \;\supset\subset\; e_2 = a)$$

holds, where a must appear neither in e_1 nor in e_2. Furthermore, the characterization of an equation by this condition is equivalent to the axiom (= 3) under the other axioms.

Proof. Assume e_1 is defined. Then we can instantiate the variable a by e_1 by ($\forall E$) below. Hence $e_1 = e_1 \supset\subset e_2 = e_1$ holds. By (= 1) and (= 2), we see $e_1 = e_2$. Similarly, we prove $e_1 = e_2$ from $E(e_2)$. Hence $e_1 = e_2$ holds by (= 3). The other direction is trivial. Let us derive (= 3) from this characterization. Assume

$E(e_1) \vee E(e_2) \supset e_1 = e_2$ holds. Assume $e_1 = a$ holds. Then e_1 is defined since a is total. So $e_1 = e_2$ holds. By the transitivity of $=$, we see $e_2 = a$ holds. Similarly, we can prove the other direction of the condition. So $e_1 = e_2$ holds. □

Note the set $\{a|e = a\}$ has at most one element, and e is defined (or total) iff it has an element. Such a set may be thought as the "partial extension" or the "partial value" of the expression e as in Fourman 1977 and Scott 1979. It would be a good exercise to derive some properties of *let* and *cond* from the axioms.

As was noted in the previous sections, we can prove Kleene's second recursion theorem from these axioms.

Proposition 2 (recursion theorem). *Let $y, x_1, \ldots, x_n$ be total variables. Assume e is an expression whose free variables are among them. Then there is a closed expression α such that $E(\alpha)$ and the following equation are derivable from the above axioms:*

$$app{*}(\alpha, x_1, \ldots, x_n) = e(\alpha, x_1, \ldots, x_n).$$

Proof. The proof is essentially the same as the proof of the second recursion theorem in Kleene 1952. The point of the proof was the construction of S^m_n functions, which freeze m variables of $m+n$-ary functions with m values. Since our Λ is an optional freezing function closure mechanism, we can define S^m_n functions. For simplicity, we assume $n = 1$. Set

$$\begin{aligned} s^1_1 &= \lambda(z, y).\Lambda(\lambda(x_1).app{*}(z, y, x_1)), \\ f &= \Lambda(\lambda(y, x_1).e(s^1_1(y, y), x_1)). \end{aligned}$$

By $(E5)$, $E(f)$ holds and f is a closed expression. Let α be the closed expression $s^1_1(f, f)$. Then, by $(beta)$ and $(E5)$, $E(\alpha)$ holds. The equation is derived as follows:

$$\begin{aligned} app{*}(\alpha, x_1) &= app{*}(\Lambda(z = f, y = f)(\lambda(x_1).app{*}(z, y, x_1)), x_1) \\ &= app{*}(f, f, x_1) \\ &= app{*}(\Lambda(\lambda(y, x_1).e(s^1_1(y, y), x_1)), f, x_1) \\ &= e(s^1_1(f, f), x_1). \end{aligned}$$ □

To show how we can define a recursive function as an application of the above proposition, we will define *append*. Let $e(y, x_1, x_2)$ be

$$cond(x_1, pair(fst(x_1), (\lambda(a, b).app{*}(y, a, b))(snd(x_1), x_2)); t, x_2).$$

Take α as in the above proposition. Let *append* be $\lambda(a, b).app{*}(\alpha, a, b)$. Then the following defining equation of *append* is provable by the above proposition.

$$append(x_1, x_2) = cond(x_1, pair(fst(x_1), append(snd(x_1), x_2)); t, x_2).$$

2.3.3. Rules for the usual logical symbols

The rules of this subsection are the more or less usual rules of logical inference. The variable condition must holds for $(\forall I)$, $(\exists E)$ below as usual. In $(\forall I)$, the variables of $\vec{v}_1, \ldots, \vec{v}_n$ must not appear in $\Gamma - \{\vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n\}$ as free variables. In $(\exists E)$, any formula of $\Pi - \{A, \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n\}$ does not have variables of $\vec{v}_1, \ldots, \vec{v}_n$ as free variables.

(*alpha*) and (*replacement*) below are considered as derived rules in the usual setting. But they are included as basic inference rules on purpose, since the realizers associated with the rules in the next section are different from the realizers of the derived rules by means of the other rules. The role of (*replacement*) is similar to the role of the consequence rule of Hoare logic and a rule of subtype concept of Constable et al. 1986. See 4.2 and 4.6.

(*assume*) $\{A\} \Rightarrow A$

$(\top)$ $\Gamma \Rightarrow \top$

$(\bot)$ $$\frac{\Gamma \Rightarrow \bot}{\Gamma \Rightarrow A}$$

(*thinning*) $$\frac{\Gamma \Rightarrow A}{\Gamma \cup \Pi \Rightarrow A}$$

(*inst*) Let $x_1, \ldots, x_m$ be total variables, $y_1, \ldots, y_n$ be class variables, $z_1, \ldots, z_p$ be partial variables, and σ be a substitution defined by

$$[e_1/x_1, \ldots, e_m/x_m, e_{m+1}/y_1, \ldots, e_{m+n}/y_n, e_{m+n+1}/z_1, \ldots, e_{m+n+p}/z_p].$$

Then the following is a logical inference:

$$\frac{\begin{array}{l}\Gamma \Rightarrow A \\ \Pi_1 \Rightarrow E(e_1)\ \ldots\ \Pi_m \Rightarrow E(e_m) \\ \Delta_{m+1} \Rightarrow Class(e_{m+1})\ \ldots\ \Delta_{m+n} Class(e_{m+n})\end{array}}{\Gamma\sigma \cup \Pi_1 \cup \ldots \cup \Pi_m \cup \Delta_{m+1} \cup \ldots \cup \Delta_{m+n} \Rightarrow A\sigma}$$

(*cut*) $$\frac{\Gamma \Rightarrow A \quad \Pi \Rightarrow B}{\Gamma \cup (\Pi - \{A\}) \Rightarrow B}$$

(*alpha*) If Γ and Π, and A and B are α-convertible to each other, respectively, then the following is an inference rule:

$$\frac{\Gamma \Rightarrow A}{\Pi \Rightarrow B}$$

(*replacement*) For any context $A[*]$ and rank 0 formulas B and C, the following is an inference rule:

$$\frac{\Gamma \Rightarrow A[B]_+ \quad \Pi \Rightarrow Env_{A[*]}[B \supset C]}{\Gamma \cup \Pi \Rightarrow A[C]_+}, \quad \frac{\Gamma \Rightarrow A[C]_- \quad \Pi \Rightarrow Env_{A[*]}[B \supset C]}{\Gamma \cup \Pi \Rightarrow A[B]_-}$$

($\wedge I$)
$$\frac{\Gamma_1 \Rightarrow A_1 \ \ldots \ \Gamma_n \Rightarrow A_n}{\Gamma_1 \cup \ldots \cup \Gamma_n \Rightarrow A_1 \wedge \ldots \wedge A_n}$$

($\wedge E$)
$$\frac{\Gamma \Rightarrow A_1 \wedge \ldots \wedge A_n}{\Gamma \Rightarrow A_{i_1} \wedge \ldots \wedge A_{i_m}} \quad (\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\})$$

($\vee I$)
$$\frac{\Gamma \Rightarrow A_i}{\Gamma \Rightarrow A_1 \vee \ldots \vee A_{i-1} \vee A_i \vee A_{i+1} \vee \ldots \vee A_n}$$

($\vee E$)
$$\frac{\Gamma \Rightarrow A_1 \vee \ldots \vee A_n \quad \Pi_1 \Rightarrow C, \ldots, \Pi_n \Rightarrow C}{\Gamma \cup \Pi_1 - \{A_1\} \cup \ldots \cup \Pi_n - \{A_n\} \Rightarrow C}$$

($\supset I$)
$$\frac{\Gamma \Rightarrow B}{\Gamma - \{A\} \Rightarrow A \supset B}$$

($\supset E$)
$$\frac{\Gamma \Rightarrow A \supset B \quad \Pi \Rightarrow A}{\Gamma \cup \Pi \Rightarrow B}$$

($\neg I$)
$$\frac{\Gamma \Rightarrow \bot}{\Gamma - \{A\} \Rightarrow \neg A}$$

($\neg E$)
$$\frac{\Gamma \Rightarrow \neg A \quad \Pi \Rightarrow A}{\Gamma \cup \Pi \Rightarrow \bot}$$

$$(\forall I) \qquad \frac{\Gamma \Rightarrow A}{\Gamma - \{\vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n\} \Rightarrow \forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A}$$

$(\forall E)$ Let $\vec{v}_i$ be a tuple variable of the form $[x_1^i, \ldots, x_{m_i}^i]$ for each $i = 1, \ldots, n$, let $y_1, \ldots, y_p$ be the free class variables of $\vec{v}_1, \ldots, \vec{v}_n$, and let σ be the substitution defined by

$$[e_1^1/x_1^1, \ldots, e_{m_1}^1/x_{m_1}^1, \ldots, e_1^n/x_1^n, \ldots, e_{m_n}^n/x_{m_n}^n].$$

Then the following is an inference rule:

$$\frac{\begin{array}{l}\Gamma \Rightarrow \forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A \\ \Pi_1 \Rightarrow [e_1^1, \ \ldots \ , e_{m_1}^1] : e_1 \ \ldots \ \Pi_n \Rightarrow [e_1^n, \ \ldots \ , e_{m_n}^n] : e_n \\ \Delta_1 \Rightarrow Class(\sigma(y_1)) \ \ldots \ \Delta_p \Rightarrow Class(\sigma(y_p))\end{array}}{\Gamma \cup \Pi_1 \cup \ldots \cup \Pi_n \cup \Delta_1 \cup \ldots \cup \Delta_p \Rightarrow A\sigma}$$

$(\exists I)$ Let $\vec{v}_1, \ldots \vec{v}_n, y_1, \ldots y_p$ and σ be the same as above. Then the following is an inference rule:

$$\frac{\begin{array}{l}\Gamma \Rightarrow A\sigma \\ \Pi_1 \Rightarrow [e_1^1, \ \ldots \ , e_{m_1}^1] : e_1 \ \ldots \ \Pi_n \Rightarrow [e_1^n, \ \ldots \ , e_{m_n}^n] : e_n \\ \Delta_1 \Rightarrow Class(\sigma(y_1)) \ \ldots \ \Delta_p \Rightarrow Class(\sigma(y_p))\end{array}}{\Gamma \cup \Pi_1 \cup \ldots \cup \Pi_n \cup \Delta_1 \cup \ldots \cup \Delta_q \Rightarrow \exists \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A}$$

$$(\exists E) \qquad \frac{\Gamma \Rightarrow \exists \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A \quad \Pi \Rightarrow C}{\Gamma \cup \Pi - \{A, \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n\} \Rightarrow C}$$

2.3.4. Rules for ∇ and $\rightarrow$

Each of the two logical symbols ∇ and $\rightarrow$ has two elimination rules and two introduction rules. Each of the eight rules corresponds to a rule of $\wedge$,$\vee$,$\forall$, or $\exists$ as indicated in the name of the rule like $(\rightarrow \vee I)$. The rules $(\nabla\exists E)$ and $(\nabla\forall I)$ have to satisfy the variable conditions. In $(\nabla\exists E)$, the variables of $p_1, \ldots, p_n$ must not appear in the following

$$\Pi - \{A, exp(p_1) = e_1, \ldots, exp(p_n) = e_n, E(e_1), \ldots, E(e_n)\} \Rightarrow C$$

as free variables. In $(\nabla\forall I)$, the variables of $p_1, \ldots, p_n$ must not appear in the sequent

$$\Gamma - \{exp(p_1) = e_1, \ldots, exp(p_n) = e_n, E(e_1), \ldots, E(e_n)\}$$

as free variables. Furthermore, free variables of $e_1, \ldots, e_n$ must not appear in the patterns $p_1, \ldots, p_n$.

Now we state the inference rules for $\to$ and ∇.

$(\to \vee I)$
$$\frac{\Gamma_1 \Rightarrow e_1 = nil \;\ldots\; \Gamma_{i-1} \Rightarrow e_{i-1} = nil \quad \Gamma_i \Rightarrow e_i : T \quad \Gamma_{i+1} \Rightarrow A_i}{\Gamma_1 \cup \ldots \cup \Gamma_{i+1} \Rightarrow e_1 \to A_1; \ldots; e_n \to A_n}$$

$(\to \vee E)$ Let S_i be $\{e_1 = nil, \ldots, e_{i_1} = nil, e_i : T\}$ for each $i = 1, \ldots, n$. Then the following is an inference rule:

$$\frac{\Gamma \Rightarrow e_1 \to A_1; \ldots; e_n \to A_n \quad \Pi_1 \Rightarrow C \ldots \Pi_n \Rightarrow C}{\Gamma \cup \Pi_1 - (\{A_1\} \cup S_1) \cup \ldots \cup \Pi_n - (\{A_n\} \cup S_n) \Rightarrow C}$$

$(\to \wedge I)$ Let S_i be the same as the above. Then the following is an inference rule:

$$\frac{\Gamma_1 \Rightarrow A_1 \;\ldots\; \Gamma_n \Rightarrow A_n \quad \Pi \Rightarrow Sor(e_1, \ldots, e_n)}{\Pi \cup \Gamma_1 - S_1 \cup \ldots \cup \Gamma_n - S_n \Rightarrow e_1 \to A_1; \ldots; e_n \to A_n}$$

$(\to \wedge E)$
$$\frac{\begin{array}{l}\Gamma \Rightarrow e_1 \to A_1; \ldots; e_n \to A_n \\ \Pi_1 \Rightarrow e_1 = nil \;\ldots\; \Pi_{i-1} \Rightarrow e_i = nil \quad \Pi_i \Rightarrow e_i : T\end{array}}{\Gamma \cup \Pi_1 \cup \ldots \cup \Pi_i \Rightarrow A_i}$$

$(\nabla\exists I)$
$$\frac{\begin{array}{l}\Gamma \Rightarrow A\sigma \\ \Pi_1 \Rightarrow exp(p_1)\sigma = e_1 \;\ldots\; \Pi_n \Rightarrow exp(p_n)\sigma = e_n \\ \Delta_1 \Rightarrow E(e_1) \;\ldots\; \Delta_n \Rightarrow E(e_n)\end{array}}{\Gamma \cup \Pi_1 \cup \ldots \cup \Pi_n \cup \Delta_1 \cup \ldots \cup \Delta_n \Rightarrow \nabla p_1 = e_1, \ldots, p_n = e_n.A}$$

$(\nabla\exists E)$
$$\frac{\Gamma \Rightarrow \nabla p_1 = e_1, \ldots, p_n = e_n.A \qquad \Pi \Rightarrow C}{\Gamma \cup (\Pi - \{A, exp(p_1) = e_1, \ldots, exp(p_n) = e_n, E(e_1), \ldots, E(e_n)\}) \Rightarrow C}$$

$(\nabla\forall I)$
$$\frac{\begin{array}{l}\Gamma \Rightarrow A \\ \Pi_1 \Rightarrow exp(p_1)\sigma = e_1 \;\ldots\; \Pi_n \Rightarrow exp(p_n)\sigma = e_n \\ \Delta_1 \Rightarrow E(e_1) \;\ldots\; \Delta_n \Rightarrow E(e_n)\end{array}}{\begin{array}{l}\Gamma - \{exp(p_1) = e_1, \ldots, exp(p_n) = e_n, E(e_1), \ldots, E(e_n)\} \\ \qquad \cup\, \Pi_1 \cup \ldots \cup \Pi_n \cup \Delta_1 \cup \ldots \cup \Delta_n \Rightarrow \nabla p_1 = e_1, \ldots, p_n = e_n.A\end{array}}$$

$(\nabla\forall E)$
$$\frac{\begin{array}{l}\Gamma \Rightarrow \nabla p_1 = e_1, \ldots, p_n = e_n.A \\ \Pi_1 \Rightarrow exp(p_1)\sigma = e_1 \;\ldots\; \Pi_n \Rightarrow exp(p_n)\sigma = e_n\end{array}}{\Gamma \cup \Pi_1 \cup \ldots \cup \Pi_n \Rightarrow A\sigma}$$

These rules characterize the conditional formula and the ∇-quantifier. Let Φ be the formula

$$\begin{array}{l} (e_1 : T \wedge A_1) \\ \qquad \vee (e_2 : T \wedge e_1 = nil \wedge A_2) \\ \qquad\qquad \vee \ldots \vee \\ \qquad\qquad\quad (e_n : T \wedge e_{n-1} = nil \wedge \ldots \wedge e_1 = nil \wedge A_n). \end{array}$$

Then, by the rules $(\to \vee I)$, $(\to \vee E)$, we can prove $e_1 \to A_1; \ldots; e_n \to A_n$ is logically equivalent to the formula Φ. So these rules gives a complete characterization of the conditional formula as a disjunctive formula. On the other hand, $(\to \wedge I)$, $(\to \wedge E)$ fail to characterize the conditional formula as conjunctive formula. The intended conjunctive meaning of the conditional formula is

$$\begin{array}{l} Sor(e_1, \ldots, e_n) \\ \quad \wedge (e_1 : T \supset A_1) \wedge (e_2 : T \supset e_1 = nil \supset A_2) \\ \qquad\qquad \wedge \ldots \wedge \\ \qquad\qquad (e_n : T \supset e_{n-1} = \ldots = e_1 = nil \supset A_n). \end{array}$$

But the above two conjunctive rules fails to imply the fact $Sor(e_1, \ldots, e_n)$ is a necessary condition of the conditional formula. But, by the aid of the disjunctive rules for $\to$, we can prove that the conditional formula is logically equivalent to the above conjunctive formula. We can derive the conjunctive rules for $\to$ by the disjunctive rules for $\to$. But it is more natural to include them as primitive rules. Note that we used $Sor(e_1, \ldots, e_n)$ in the premises of $(\to \wedge I)$. Since the "serial or" Sor itself is a conditional formula, we may apply the rules of $\to$ to it.

The situation is completely the same in the case of the ∇-quantifier. Our intended existential representation and universal representation of the formula $\nabla p_1 = e_1, \ldots, x_n = e_n.A$ are as follows:

$$\exists \vec{x}.(exp(p_1) = e_1 \wedge \ldots \wedge exp(p_n) = e_n \wedge A),$$

$$\begin{array}{l} \Diamond \exists \vec{x}.(exp(p_1) = e_1 \wedge \ldots \wedge exp(p_n) = e_n) \\ \qquad\qquad \wedge \forall \vec{x}.(exp(p_1) = e_1 \supset \ldots \supset exp(p_n) = e_n \supset A), \end{array}$$

where $\vec{x}$ is the sequence of variables $FV(p_1) \cup \ldots \cup FV(p_n)$.

The following lemma will be used to prove the soundness theorem of **px**-realizability in 3.1.

Lemma 1. *Assume a formula A is strict with respect to a variable x, i.e., $A[e/x] \Rightarrow E(e)$ holds. Then the following holds in* **PX***:*

(1) $A[let\ p = e_1\ in\ e_2/x]$ *iff* $\nabla p = e_1.A[e_2/x]$,
(2) $A[cond(e_1, d_1; \ldots; e_n, d_n)/x]$ *iff* $e_1 \to A[d_1/x]; \ldots; e_n \to A[d_n/x]$,
(3) When x *does not belong to* $FV(e_i)$ *for* $i = 1, \ldots, n$,

$$(e_1 \to A_1; \ldots; e_n \to A_n)[cond(e_1, d_1; \ldots; e_n, d_n)/x]$$

iff

$$e_1 \to A_1[d_1/x]; \ldots; e_n \to A_n[d_n/x].$$

Proof. (1) Assume $A[let\ p = e_1\ in\ e_2/x]$ holds. By the strictness condition, we see $E(let\ p = e_1\ in\ e_2)$. By renaming bound variables, we may assume the pattern p does not have a free occurrence x, so $exp(p)[e_2/x] = exp(p)$. Hence, by substituting $let\ p = e_1\ in\ e_2$ for a of (let), we see $\nabla p = e_1.(e_2 = let\ p = e_1\ in\ e_2)$ holds. Hence, by $(\nabla\forall E)$ and $(= 4)$, we can derive

$$\{A[let\ p = e_1\ in\ e_2/x], exp(p) = e_1\} \Rightarrow A[e_2/x].$$

By $(\nabla\forall I)$, the only-if part of (1) is derived from this sequent. The variable conditions may be assumed to hold by renaming variables of the patterns by the aid of $(alpha)$. The if part is similarly proved.

(2) Assume $A[cond(e_1, d_1; \ldots; e_n, d_n)/x]$ holds. Then

$$E(cond(e_1, d_1; \ldots; e_n, d_n))$$

holds. Hence, by substituting $cond(e_1, d_1; \ldots; e_n, d_n)$ for a of $(cond)$, we see

$$e_1 \to cond(e_1, d_1; \ldots; e_n, d_n) = d_1; \ldots; e_n \to cond(e_1, d_1; \ldots; e_n, d_n) = d_n$$

holds. If $e_1 = nil, \ldots, e_{i-1} = nil, e_i : T$ hold, then, by $(\to \wedge E)$,

$$cond(e_1, d_1; \ldots; e_n, d_n) = d_i$$

holds, hence $A[d_i/x]$ holds. By $(\to \vee I)$, we see

$$e_1 \to A[d_1/x]; \ldots; e_n \to A[d_n/x].$$

Hence, by $(\to \vee E)$, we see the only-if part of (2) holds. The if part is similarly proved.

(3) This is proved from (2) by the aid of the rules for $\to$. □

2.3.5. Axioms for the modal operator

($\Diamond$1) $$\neg\neg A \supset\subset \Diamond A$$

($\Diamond$2) $$\forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.\Diamond F \supset \Diamond\forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.F$$

($\Diamond$3) $$A \supset\subset \Diamond A \quad (\ A \text{ is a rank 0 formula})$$

The modal operator $\Diamond$ is just double negation. So ($\Diamond$2) is the principle of double negation shift (Troelstra 1973). By these axioms, we see that if we restrict formulas to rank 0 formulas, then the logic of **PX** is *classical*, i.e., the following proposition holds.

Proposition 3. *A sequent $\Gamma \Rightarrow A$ is provable in* **PX**+*(classical logic) iff $\Gamma \Rightarrow \Diamond A$ is provable in* **PX**. *Especially,* **PX**+*(classical logic) is a conservative extension of* **PX** *over rank 0 formulas.*

Proof. Assume $\Gamma \Rightarrow A$ is provable in **PX**+(classical logic). This means $\Gamma \Rightarrow A$ is provable from the additional initial sequents of the form $\{\ \} \Rightarrow F \vee \neg F$ and the rules and axioms of **PX**. Then there are formulas $A_1, \ldots, A_n$ such that $\Gamma \cup \{\forall \vec{x}_1.(A_1 \vee \neg A_1), \ldots, \forall \vec{x}_n.(A_1 \vee \neg A_n)\} \Rightarrow A$ is provable in **PX**, where $\vec{x}_i$ is the sequence of all total variables of A_i. This is not so trivial as in the usual logic, since $\{\ \} \Rightarrow F \vee \neg F$ is not always equivalent to $\{\ \} \Rightarrow \forall \vec{x}(F \vee \neg F)$ for F may have free partial variables which cannot be quantified. But this is proved by a straightforward induction on the length of the proof by the fact that all eigenvariables of the rules with variable conditions, i.e., $(\forall I)$, $(\exists E)$, $(\nabla\forall I)$, and $(\nabla\exists E)$, are total variables. We leave the proof to the reader.

In general, if $\Gamma \cup \Pi \Rightarrow A$ is provable, then $\Gamma \cup \{\neg\neg B | B \in \Pi\} \Rightarrow \neg\neg A$ is derivable from it only by $(\neg I)$ and $(\neg E)$. On the other hand, the theorem $\forall \vec{x}_i.\neg\neg(A_i \vee \neg A_i)$ is provable in intuitionistic predicate logic.

By ($\Diamond$1) and ($\Diamond$2), we can derive $\neg\neg\forall \vec{x}_i.(A_i \vee \neg A_i)$. Hence, we see $\Gamma \Rightarrow \Diamond A$ is provable. The other direction is trivial. By ($\Diamond$3), **PX**+(classical logic) is a conservative extension of **PX** over rank 0 formulas. (Note that ($\Diamond$2) is equivalent to $\neg\neg\forall \vec{x}(A \vee \neg A)$.) $\square$

By the above proposition, we may prove rank 0 formulas by classical reasoning. This means that parts of proofs concerning rank 0 formulas do not have any effects on the programs extracted from the proofs. In actual proof building, we often assume rank 0 lemmas without proof, if their validity is mathematically clear.

In the actual implementation, we do not have the above axioms, but we have a tautology checker which checks if a rank 0 formula is derivable from a subsystem of classical first order logic. Although this is essentially a tautology checker of classical propositional logic, the above axioms are proved by it (see 7.2.4). These axioms can also be proved with the aid of the EKL translator (see 7.4).

As an application of proposition 3, we will show that **PX** can derive a form of Markov's principle (see Troelstra 1973). Computer scientists call the principle "Dijkstra's linear search". It reads

If one has a procedure deciding if $P(n)$ holds for each $n \in N$ and one knows $\neg\neg\exists x \in N.P(n)$, then one can find $n_0 \in N$ such that $P(n_0)$.

A logical formulation of the principle is

(MR) $$\forall n \in N.(P(n) \vee \neg P(n)) \wedge \neg\neg\exists n \in N.P(n) \supset \exists n \in N.P(n).$$

We prove a weaker version of (MR). Let f be a function of **PX** such that

(A) $$\mathbf{PX} \vdash \forall x : N, y : N.E(f(x,y)).$$

Then Markov's principle for f is stated as

(MR$_f$) $$\{x : N, \Diamond\exists y : N.f(x,y) : T\} \Rightarrow \exists y : N.f(x,y) : T.$$

Define functions min_f, g by

$$min_f(x) = g(x,0), \quad g(x,y) = cond(f(x,y), y; t, g(x,y+1)).$$

Resorting classical logic and ($\Diamond$1), we can prove the following by (A):

$$\{x : N, \Diamond\exists y : N.f(x,y) : T\} \Rightarrow \nabla y = min_f(x).f(x,y) : T.$$

Since the conclusion of this sequent is of rank 0, this is provable in **PX** by proposition 3 and ($\Diamond$3), and (MR_f) is derivable from this.

2.3.6. Miscellaneous axioms

The axioms of this subsection are not so essential. Some of them are even optional, and others depend on the DEF system on which **PX** is based (see 7.2.6). The actual **PX** system has a data base of minute axioms with query functions by which a user can ask the system if his intended formulas belong to the data base. All axioms in this subsection except (def), $(N5)$, $(V4)$ can be *proved* from the axioms of the data base. (These three axioms are *generated* by other functions.) In this subsection α,β,γ,… are metavariables for objects, and a,a_1,…,b,b_1,… are *total* individual variables.

($constant$)

$$\begin{gathered}
quote(\alpha) = quote(\beta) \qquad (\alpha = \beta),\\
\neg quote(\alpha) = quote(\beta) \qquad (\alpha \neq \beta),\\
t = quote(t), \quad nil = quote(nil), \quad 0 = quote(0),\\
quote(\alpha) : Atm \qquad (\alpha \in Atom),\\
quote((\alpha\ .\ \beta)) = pair(quote(\alpha), quote(\beta)),\\
quote(\nu + 1) = suc(quote(\nu)) \quad (\nu \in N).
\end{gathered}$$

(def) For each definition $f_1(\vec{v}_1) = e_1, \ldots, f_n(\vec{v}_n) = e_n$ of the DEF system, the following are axioms:

$$f_1(\vec{v}_1) = e_1, \ldots, f_n(\vec{v}_n) = e_n$$

($totality$) $\quad E(atom(a)),\ E(equal(a,b)),\ E(pair(a,b)),\ E(list(a_1,\ldots,a_n))$

($char$) $\quad atom(a) = t \supset\subset a : Atm,\ equal(a,b) = t \supset\subset a = b,\ a : T \supset\subset \neg a = nil$

($N1$) $$0 : N, \quad \forall a : N.suc(a) : N$$

($N2$) $$\forall a : N.prd(suc(a)) = a, \quad \forall a : N.(\neg a = 0 \supset\subset suc(prd(a)) = a)$$

($N3$) $$\forall X.(0 : X \wedge \forall a : X.suc(a) : X \supset \forall a : N.a : X)$$

($V1$) $$E(e) \supset e : V$$

($V2$) $$\begin{gathered} fst(pair(a,b)) = a, \quad snd(pair(a,b)) = b,\\ \neg a : Atm \supset\subset pair(fst(a), snd(a)) = a \end{gathered}$$

($V4$) $$\forall X.(\forall a : Atm.a : X \wedge \forall a : X, b : X.pair(a,b) : X \supset \forall a : V.a : X).$$

2.4. Classes: data types as objects

A *class* is a *code* of a set of objects. Classes and axioms on them play very important roles in **PX** in two ways. For one thing, we can define various kinds of data types as classes including dependent types in the sense of Martin-Löf. Since a class is a code of a set, we can program various operations on types. So data types are first class objects and the type discipline of **PX** is close to those of the typed languages Russell and Pebble, although it does not have the type of all types. For another, the induction principles for the inductively defined classes are quite useful for representations of recursions called CIG recursions and their domains of terminations are also representable by classes. Besides these, class is useful in characterizing the values of expressions and graphs of functions. In 2.4.3, we will show how to axiomatize the minimality of the graph of a recursively defined function by using the induction principle of CIG inductive definition.

The axioms on classes are divided into two groups. One group contains the axioms of conditional inductive generation (CIG), which maintains the existence of a particular kind of inductively defined sets. The other contains the axioms of dependent types, which maintains the existence of dependent types in the sense of Martin-Löf. The former is described in 2.4.1, and the latter is described in 2.4.2.

2.4.1. CIG: conditional inductive generation

CIG is a generalization of inductive generation (IG) of Feferman 1979. CIG plays many important roles in **PX**. In fact, CIG is the "heart" of **PX**. For one thing, CIG is a method by which we can define various classes, including Cartesian products, function spaces, and various recursive data types. For another, it is a device for representing fairly wide groups of recursions through proof rules associated with recursively defined classes (see chapter 4). In fact, we can derive *all* recursive functions in **PX** via CIG. It also provides a way to formulate our version of fixed point induction (see 2.4.3).

The principle of CIG is quite similar to recursive definitions of types and their rules of Nuprl (Constable and Mendler 1985). Although their recursively defined types also include reflexive domains, induction rules are given only to the types defined by the positive recursive definition forms. So the proof rules of Nuprl are very closed to the proof rules of CIG.

First, we will give a simplified formulation of CIG, to which we will give a mathematical semantics in 6.3. Later we will give a more realistic formulation. In the following, variables $\vec{X} = X_0, \ldots, X_n$ are always class variables.

Definition 1 (CIG templates). Formulas are called *CIG templates with re-*

spect to $\vec{X}$ and n_0, $CIG_{n_0}(\vec{X})$ for short, are generated by the following grammar:

$$\begin{aligned}
H ::=& E(e)|PC|e = e|\top|\bot \\
&|H \wedge \ldots \wedge H|e \to H; \ldots ; e \to H|\Diamond P|K \supset H|\neg K \\
&|\forall CV, \ldots , CV.H|\nabla p = e, \ldots , p = e.H \\
\\
P ::=& H|P \wedge \ldots \wedge P|e \to P; \ldots ; e \to P|K \supset P|P \vee \ldots \vee P \\
&|\forall CV, \ldots , CV.P|\exists CV, \ldots , CV.P|\nabla p = e, \ldots , p = e.P \\
\\
K ::=& E(e)|KC|e = e|\top|\bot \\
&|K \wedge \ldots \wedge K|K \vee \ldots \vee K|e \to K; \ldots ; e \to K|\Diamond K|P \supset K|\neg P \\
&|\forall CV, \ldots , CV.K|\exists CV, \ldots , CV.K|\nabla p = e, \ldots , p = e.K \\
PC ::=& [e_1, \ldots , e_{n_0}] : X_0|[e, \ldots , e] : X_1| \ldots |[e, \ldots , e] : X_n|[e, \ldots , e] : C \\
\\
KC ::=& [e, \ldots , e] : X_1| \ldots |[e, \ldots , e] : X_n|[e, \ldots , e] : C \\
\\
CV ::=& \vec{x} : X_1| \ldots |\vec{x} : X_n|\vec{x} : C
\end{aligned}$$

where H ranges over $CIG_{n_0}(\vec{X})$, $e, e_1, ..$ range over expressions in which X_0 does not occur, p ranges over patterns, C ranges over class constants, and $\vec{x}$ ranges over tuples of individual variables. Note that the length of the argument tuple of X_0 *must* be n_0. P (K) ranges over formulas in which $[e_1, \ldots , e_{n_0}] : X_0$ appears only positively (negatively).

It is possible to extend the definition of CV so that a formula like

$$\Diamond \exists x_1, \ldots , x_{n_0} : X_0.H$$

is a CIG template, too. But we do not do so, in order to avoid the complication of substituting a formula for X_0. Instead, it must be written as

$$\Diamond \exists x_1, \ldots , x_{n_0}.([x_1, \ldots , x_{n_0}] : X_0 \wedge H).$$

This definition of CIG templates is not very useful to humans. The following is an equivalent and more recognizable definition.

Definition 2 (another definition of CIG templates). A formula H is a CIG template of $CIG_{n_0}(\vec{X})$ iff it satisfies the conditions:

(1) H is a rank 0 formula.
(2) If a subformula of H has the form $[e_1, \dots, e_n] : e_{n+1}$, then e_{n+1} must be a class constant or belong to $\vec{X}$ and $e_1, \dots, e_n$ have no occurrence of X_0. Furthermore, if e_{n+1} is X_0, then the entire subformula must occur in a positive part of H and n is n_0. If a subformula of H has the form $E(e)$, then e has no occurrence of X_0. If a subformula of H has the form $e_1 = e_2$, then e_1 and e_2 have no occurrence of X_0.
(3) For any subformula of the form $\forall \vec{x}_1 : e_1, \dots, \vec{x}_n : e_n.F$, $\vec{x}_1, \dots, \vec{x}_n$ are tuples of *individual* variables and e_i is an element of $\vec{X}$ or a class constant. Similarly for the existential quantifier and the ∇-quantifier.
(4) H has no occurrence of the predicate symbol $Class$.

The important point is that the condition (1) maintains every CIG template is a rank 0 formula. It is possible to drop the restriction that CIG templates are of rank 0 (see Feferman 1979 and Tatsuta 1987). We put the restriction so that classes have no hidden information. See 2.6.2 for a discussion.

For each $A \in CIG_n(\vec{X})$ and variables $\vec{a} = a_1, \dots, a_n$, we assume that there is an expression $\mu X_0\{[\vec{a}]|A\}$, whose free variables are just $FV(A) - \{X_0, a_1, \dots, a_n\}$. This expression is called a *class expression*. We will often abbreviate $\mu X_0\{[\vec{a}]|A\}$ to $\mu X_0\{\vec{a}|A\}$. For example, $\mu X_0\{[a_1, a_2]|A\}$ will be abbreviated to $\mu X_0\{a_1, a_2|A\}$. The class expression stands for the smallest fixed point of the monotone map "$X_0 \mapsto \{\vec{a}|A\}$". So the following are axioms:

$(CIG\ def)$

$$Class(\mu X_0\{\vec{a}|A\}), \quad \nabla[a_1, \dots, a_n] = x.A[\mu X_0\{\vec{a}|A\}/X_0] \supset\subset x : \mu X_0\{\vec{a}|A\}.$$

If X_0 does not appear in A, we write $\{\vec{a}|A\}$. Then this class notation is the same as the usual set notation. The inductive definition of $\mu X_0\{\vec{a}|A\}$ is called *CIG inductive definition*. The induction rule for the inductive definition of $\mu X_0\{\vec{a}|A\}$, called *CIG induction* is

$(CIG\ ind)$

$$\frac{\Gamma \Rightarrow F(\vec{a})}{\{[\vec{a}] : \mu X_0\{\vec{a}|A\}\} \cup \Gamma - \{A[F(\vec{a})/X_0], A[\mu X_0\{\vec{a}|A\}/X_0]\} \Rightarrow F(\vec{a})},$$

where the variables of $\vec{a}$ must not be free variables of

$$\Gamma - \{A[F(\vec{a})/X_0], A[\mu X_0\{\vec{a}|A\}/X_0]\}.$$

We substituted a *formula* $F(\vec{a})$ for a class variable X_0. This means replacing all of the subformula of the form $[e_0, \dots, e_n] : X_0$ by $F[e_0/a_1, \dots, e_n/a_n]$ and substituting the class expression $\mu X_0\{\vec{a}|A\}$ for the other occurrences of X_0.

Note that we have not introduced an expression constructor for the class notation $\mu X_0\{\vec{a}|A\}$. If we introduced such a constructor, then we would have to define expressions and formulas simultaneously. It is possible to conceive of the introduction of such a constructor as natural; in fact, Feferman and Martin-Löf took this approach. But we take another approach. Let Φ be the set

$$\{\langle \vec{a}, \vec{v}, \vec{X}, A\rangle | A \in CIG_n(\vec{X}) \wedge \{v_1, \ldots, v_m\} = FV(A) - \{X_0, a_1, \ldots, a_n\}\},$$

where $\vec{v}$ is the tuple of variables $v_1, \ldots, v_m$. Since Φ is countably infinite, there are one-to-one mappings from the set into the set

$$B - \{app, app{*}, list, atom, fst, snd, pair, equal, suc, prd\}.$$

(Recall that B is a countably infinite set.) Fix one such mappings, say *cigname*. Then the class expression $\mu X_0\{\vec{a}|A\}$ for $A \in CIG_n(\vec{X})$ is

$$(cigname(\langle \vec{a}, \vec{v}, \vec{X}, A\rangle))(v_1, \ldots, v_m),$$

where $\vec{v} = v_1, \ldots, v_m$. (We assume *cigname* is chosen such that the arity of $cigname(\langle \vec{a}, \vec{v}, \vec{X}, A\rangle)$ is m.) The function names belonging to $cigname(\Phi)$ are called *class functions*. Note that a function name $cigname(\langle \vec{a}, \vec{v}, \vec{X}, A\rangle)$ may occur in the CIG template A. In the implementation of **PX**, users can declare a function identifier for a name of $cigname(\langle \vec{a}, \vec{v}, \vec{X}, A\rangle)$ for each CIG template A. This reflects the above formal treatment of class functions.

CIG in the implementation of **PX** described in chapter 7 is more flexible than the version of CIG presented here in respect to the following:

(0) Simultaneous definitions of classes are available.
(1) If e is an expression whose class variables belong to $X_1, \ldots, X_n$ (X_0 not allowed) and is confirmed to be a class by a specific algorithm, then not only $[e_1, \ldots, e_n] : X_{i+1}$ but also $[e_1, \ldots, e_n] : e$ is allowed as H and K in the grammar of CIG templates.
(2) For a class whose body of definition is a conditional formula, a more convenient form of CIG induction is available.
(3) A class may be specified as a superclass of the class to be defined.

In the implementation of **PX**, classes can be defined through the declarations as the definition of functions in Lisp systems. Let us explain the extended CIG in terms of a declaration. For each $i = 1, \ldots, n$, let C_i be an expression of the form $f_i(\vec{a}_i, \vec{A}_i)$, where f_i is the name of the class-valued function which is going to be defined. $\vec{a}_i$ is a list of individual variables and $\vec{A}_i$ is a list of class variables. Note that both $\vec{a}_i$ and $\vec{A}_i$ may be empty lists. Then f_i is a name of a class rather than of a function. We assume $f_1, \ldots, f_n$ are each different from one another. Let $\vec{x}_i$ $(i = 1, \ldots, n)$ be a nonempty sequence of mutually different total individual variables that do not appear in C_i. Then a *generalized CIG template* with respect to $\vec{x}_1 : C_1, \ldots, \vec{x}_n : C_n$ is defined by the grammar of $CIG_{n_0}(\vec{X})$ that is modified as follows:

$$PC ::= [e_1, \ldots, e_{l_i}] : C_{l_i} | [e, \ldots, e] : \Theta$$
$$KC ::= [e, \ldots, e] : \Theta$$
$$CV ::= \vec{x} : \Theta,$$

where l_i is the length of $\vec{x}_i$ and Θ is a *class expression* over $\vec{A}_1, \ldots, \vec{A}_n$ that is defined by

1. A class constant is a class expression over $\vec{A}_1, \ldots, \vec{A}_n$.
2. Class variables $\vec{A}_1, ..., \vec{A}_n$ are class expressions over $\vec{A}_1, \ldots, \vec{A}_n$.
3. When $g(b_1, \ldots, b_{m_0}, B_1, \ldots, B_{m_1})$ is a declared CIG definiens (see below), $e_1, \ldots, e_{m_0}$ is a list of expressions which have been certified to have values, and $\Theta_1, \ldots, \Theta_{m_1}$ is a list of class expressions over $\vec{A}_1, \ldots, \vec{A}_n$. Then $g(e_1, \ldots, e_{m_0}, \Theta_1, \ldots, \Theta_{m_1})$ is a class expression. Note that the CIG definiens $g(b_1, \ldots, b_{m_0}, B_1, \ldots, B_{m_1})$ *must have been declared*, so it must not be one of the f_i that is going to be defined.

Then a declaration of $C_1, \ldots, C_n$ by a simultaneous CIG inductive definition takes the form

$$(CIG\ dec) \qquad \begin{aligned} \textbf{deCIG}\ \vec{x}_1 : C_1 &\equiv_{D_1} body_1 \\ \vec{x}_2 : C_2 &\equiv_{D_2} body_2 \\ &\ldots \\ \vec{x}_n : C_n &\equiv_{D_n} body_n. \end{aligned}$$

In this definition,

$$\begin{aligned} body &::= clause \mid clause\ ,\ body \\ clause &::= expression\ \rightarrow\ CIG\text{-}templatelist. \end{aligned}$$

CIG-templatelist stands for a *non-empty* list of CIG templates and $D_1, \ldots, D_n$ are class expressions over $\vec{A}_1, \ldots, \vec{A}_n$. $C_1, \ldots, C_n$ are called *CIG definiens* of this declaration. The bodies of the definition must satisfy the variable condition: all of free variables of $body_i$ and D_i must appear among free variables of $\vec{x}_i : C_i$.

Let $body_i$ be

$$e_1^i\ \rightarrow\ \phi_{1,1}^i, \ldots, \phi_{1,q_1^i}^i\ , \ldots,\ e_{p_i}^i\ \rightarrow\ \phi_{p_i,1}^i, \ldots, \phi_{p_i,q_{p_i}^i}^i.$$

After the declaration of the classes is made, the following become axioms of **PX**:

$$Class(C_i) \quad (i{=}1,\ldots,n)$$

$$\vec{x}_i : C_i \supset\subset \vec{x}_i : D_i \wedge e_1^i \to \bigwedge_{j=1}^{q_1^i} \phi_{1,j}^1; \ldots ; e_{p_i}^i, \to \bigwedge_{j=1}^{q_{p_i}^i} \phi_{p_i,j}^i \quad (i{=}1,\ldots,n).$$

The induction principle for the classes is

$$(CIG\ ind^*) \qquad \frac{\Gamma_1^1 \Rightarrow F_1 \ \ldots\ \Gamma_{p_1}^1 \Rightarrow F_1 \ \ldots\ \Gamma_1^n \Rightarrow F_n \ \ldots\ \Gamma_{p_n}^n \Rightarrow F_n}{\bigcup_{i=1}^n \bigcup_{j=1}^{p_i} \widetilde{\Gamma_j^i} \Rightarrow \bigwedge_{i=1}^n (\vec{x}_i : C_i \supset F_i)},$$

where $\widetilde{\Gamma_j^i}$ is the set

$$\begin{aligned}\Gamma_j^i - (\{\vec{x}_i : D_i\} & \\ & \cup \{e_1^i = nil, \ldots, e_{j-1}^i = nil, e_j^i : T\} \\ & \cup \{\phi_{j,1}^i[F_1/C_1, \ldots, F_n/C_n], \ldots, \phi_{j,q_j^i}^i[F_1/C_1, \ldots, F_n/C_n]\} \\ & \cup \{\phi_{j,1}^i, \ldots, \phi_{j,q_j^i}^i\}),\end{aligned}$$

where the substitution $[F_1/C_1, \ldots, F_n/C_n]$ means replacement of every subformula $[e_1, \ldots, e_m] : C_i$ by the formula $F_i[e_1, \ldots, e_m/\vec{x}_i]$.

We next give some examples of classes generated by CIG. For simplicity and readability, we will use infix notations and the class (set) notation.

(1) Function space: The space of functions (programs) from X to Y is given by

$$X \to Y = \{f | \forall a : X.app{*}(f, a) : Y\}.$$

This is a space of unary functions; function spaces with more arguments are defined similarly. Note that the functions of this space are *intensional*. Even if X and Y are equipped with equivalence relation R_X and R_Y, functions of the space need not preserve them. If $\{a, b | a : X \wedge b : X \wedge R_X(a, b)\}$ and $\{a, b | a : Y \wedge b : Y \wedge R_Y(a, b)\}$ are classes, say E_X and E_Y respectively, then the extensional function space can be defined by

$$X \to Y = \{f | \forall [a, b] : E_X.[app{*}(f, a), app{*}(f, b)] : E_Y\}.$$

Then the extensional equality on the space is defined in the obvious way, and it again defines a class. So we can construct higher order extensional function spaces over it. We will see an application of such a construction of a hierarchy of extensional function spaces in 5.2. The partial function space from X to Y is also definable:

$$X \rightharpoonup Y = \{f | \forall a : X.(E(app{*}(f,a)) \supset app{*}(f,a) : Y)\}.$$

(2) Finite set: The enumeration type of Pascal or finite set is easily defined through "serial or" as follows:

$$\{x_1, \dots, x_n\} = \{a | Sor(equal(x_1, a), \dots, equal(x_n, a))\}.$$

For example, **Bool** is defined by $\{t, nil\}$. Note that $Sor()$ is $\perp$ by definition. So $\{\} = \{a|\perp\}$ is an empty class.

(3) Cartesian product: The Cartesian product of two classes is defined by

$$X \times Y = \{a_1, a_2 | a_1 : X \wedge a_2 : Y\}.$$

Similarly, we can define n-times products. The 0-times product is $\{[]|\top\}$, i.e., the class $\{nil\}$. So the lifting of a type X in the sense of Plotkin 1985 is defined as $\{nil\} \rightharpoonup X$.

(4) Disjoint sum: The disjoint sum or coproduct of two classes is defined by

$$X + Y = \{x | \nabla(a\ .\ b) = x.(equal(a,t) \to b : X; equal(a, nil) \to b : Y)\}.$$

The *fst* part of the dotted pair specifies the class to which the *snd* part belongs. Finite disjoint sum of any numbers of classes are defined similarly. The infinite disjoint sum will be introduced by the *Join* operator below.

(5) Propositional equality: The following is an implementation of the type of propositional equality of Martin-Löf 1982. Its extension equals the extension of the 0-times product $\{nil\}$ when $a = b : X$, and is empty otherwise. Martin-Löf uses a constant **r** instead of *nil*.

$$I(X, a, b) = \{x | x = nil \wedge a = b \wedge a : X \wedge b : X\}.$$

(6) Lists: The class of lists $List(X_1)$ is defined by CIG as

$$\mu X_0\{a | atom(a) \to a = nil; t \to fst(a) : X_1 \wedge snd(a) : X_0\}.$$

Then the induction principle for $List(X)$ is

$$(List1) \qquad \frac{\Gamma \Rightarrow A(a)}{\{a : List(X)\} \cup \Gamma - \{HYP1, HYP2\} \Rightarrow A(a)},$$

where

$$\begin{aligned} HYP1 &\equiv atom(a) \to a = nil; t \to fst(a) : X \wedge A(snd(a)), \\ HYP2 &\equiv atom(a) \to a = nil; t \to fst(a) : X \wedge snd(a) : List(X). \end{aligned}$$

By using the rules of conditional forms, we see that this is logically equivalent to the induction principle

$$(List2) \qquad \frac{\Gamma \Rightarrow A(a) \qquad \Pi \Rightarrow A(a)}{\left(\begin{array}{l} \{\ a : List(X)\ \} \\ \quad \cup\Gamma - \{\ atom(a) : T, a = nil\ \} \\ \qquad \cup\Pi - \left\{ \begin{array}{ll} atom(a) = nil, & fst(a) : X, \\ snd(a) : List(X), & A(snd(a)) \end{array} \right\} \end{array} \right) \Rightarrow A(a)}.$$

By means of $(CIG\ dec)$, a function $List$ can be declared as follows:

$$\begin{aligned} \textbf{deCIG}\ a : List(X) \equiv & atom(a) \to a = nil, \\ & t \to fst(a) : X, snd(a) : List(X). \end{aligned}$$

Then the induction principle for it is $(List2)$ rather than $(List1)$. In general, $(List2)$ is preferable to $(List1)$, since it is simpler and natural. In fact a simple-minded extractor will extract a list-recursion program with some redundancies from proofs using $(List1)$, since some rules on conditional formulas are involved in such proofs. On the contrary, the usual list-recursion scheme will be extracted from proofs using $(List2)$. However, the actually implemented extractor described in 3.2 extracts the same code from the proofs using these two different rules, thanks to optimizations. (See 3.2 and appendix C.)

$List(X)$ was the class of list including nil. Let us define the class of nonempty lists. The superclass feature is convenient for this. Set

$$\begin{gathered} a : Dp = \{a | \nabla(x\ .\ y) = a.\top\} \\ \textbf{deCIG}\ a : List_1(X) \equiv_{Dp} snd(a) \to fst(a) : X, snd(a) : List_1(X), \\ t \to fst(a) : X. \end{gathered}$$

The class defined by this is

$$\text{(A)} \qquad \mu X_0\{a | a : Dp \wedge (snd(a) \to fst(a) : X \wedge snd(a) : X_0; t \to fst(a) : X)\}.$$

Dp is the class of dotted pairs and it is to be a superset of $List_1(X)$ so that all lists of the class are not nil. The induction principle ($CIG\ ind$) for $List_1(X)$ is

$$(List3)\qquad \frac{\Gamma \Rightarrow A(a) \qquad \Pi \Rightarrow A(a)}{\begin{array}{l}\{\ a : List_1(X)\ \} \\ \quad \cup\left(\Gamma - \left\{\begin{array}{l} a : Dp,\ snd(a) : T,\ fst(a) : X, \\ snd(a) : List_1(X),\ A(snd(a)) \end{array}\right\}\right) \qquad \Rightarrow A(a) \\ \quad \cup(\Pi - \{\ a : Dp,\ snd(a) = nil,\ fst(a) : X\ \})\end{array}}.$$

(7) Natural numbers and S-expressions: There are a built-in classes N and V, but we cannot use full induction for them, since their induction principles ($N3$) and ($V4$) are restricted to *classes*. We cannot state an arbitrary formula for the class variable X of the axioms. So we should define a class Nat as follows:

$$\begin{aligned}\textbf{deCIG}\ a : Nat \equiv & equal(0, a) \to \top, \\ & t \to prd(a) : Nat.\end{aligned}$$

Then the usual mathematical induction is the induction for Nat. On the other hand, it is easy to prove N and Nat are extensionally equal, i.e., $\forall x.x : N \supset\subset x : Nat$. By the aid of the rule of ($replacement$), we may replace $e : N$ by $e : Nat$. So we may use the usual mathematical induction for N, or we may think of Nat as N. The same technique is applied to the class V of S-expressions. In the implementation of **PX**, N and V are treated in a special way so that the full inductions are assigned in advance.

(8) Type of dependent conditional: The typing of a dependent conditional expression like $if\ x\ then\ 3\ else\ pair(a, b)$ is a controversial subject (Burstall and Lampson 1984, Cardelli 1986). The conventional solution of this problem is to assume that two expressions after $then$ and $else$ have the same type, since the typing of the general form of such an expression involves logical inference. **PX** is a logical system, so there is no reason to avoid the type (class) of such an expression. Define

$$Cond(e_1, X_1; \ldots; e_n, X_n) = \{a | e_1 \to a : X_1; \ldots; e_n \to a : X_n\}.$$

By lemma 1 of 2.3.4, we can derive the dependent conditional typing condition

$$\begin{aligned}cond(e_1, d_1; \ldots; e_n, d_n) : Cond(e_1, X_1; \ldots; e_n, X_n)& \\ \supset\subset e_1 \to d_1 : X_1; \ldots; e_n \to d_n : X_n&.\end{aligned}$$

(9) Tree ordinals: Well-founded trees which have elements of X_1 as leaves and branch 1 or $\aleph_0$ times at each node can be defined as follows:

$$\begin{aligned}\textbf{deCIG } x : O(X_1) \equiv & \mathit{equal}(\mathit{fst}(x), 0) \to \nabla(a\ b) = x.b : X_1,\\ & \mathit{equal}(\mathit{fst}(x), 1) \to \nabla(a\ b) = x.b : O(X_1),\\ & \mathit{equal}(\mathit{fst}(x), 2) \to \nabla(a\ b) = x.\forall n : N.\mathit{app}{*}(b, n) : O(X_1)\end{aligned}$$

By iteration of this construction to N, we get the hierarchy of constructive tree ordinals $O(N), O(O(N)), \ldots$ as in Feferman 1979.

(10) Subexpression relation: The modal operator $\Diamond$ provides an easy way to use disjunction and existential quantifiers in CIG templates. For example, we can define a class Sub so that $[x, y] : \mathit{Sub}$ iff x is a sub S-expression of y:

$$\begin{aligned}\textbf{deCIG } [x, y] : \mathit{Sub} \equiv & \mathit{equal}(x, y) \to \top,\\ & \mathit{atom}(y) \to \bot,\\ & t \to \nabla(a\ .\ b) = y.\Diamond([x, a] : \mathit{Sub} \vee [x, b] : \mathit{Sub}).\end{aligned}$$

One cannot infer $[x, \mathit{fst}(y)] : \mathit{Sub} \vee [x, \mathit{snd}(y)] : \mathit{Sub}$ from $[x, y] : \mathit{Sub}$ immediately from the definition, since the body of the definition is "classicalized" by putting $\Diamond$ in front of the disjunction in the third clause. One must resort to structural induction on S-expressions to prove the inference. The following definition of subexpression relation Sub_1 with extra information r allows one to infer if x is in the *fst* part or *snd* part of y from the information r:

$$\begin{aligned}\textbf{deCIG } [r, x, y] : \mathit{Sub}_1 \equiv & \mathit{equal}(x, y) \to r = 0,\\ & \mathit{atom}(y) \to \bot,\\ & \mathit{equal}(\mathit{fst}(r), 1) \to [\mathit{snd}(r), x, \mathit{fst}(y)] : \mathit{Sub}_1,\\ & \mathit{equal}(\mathit{fst}(r), 2) \to [\mathit{snd}(r), x, \mathit{snd}(y)] : \mathit{Sub}_1.\end{aligned}$$

Then r which satisfies $[r, x, y] : \mathit{Sub}_1$ represents where x is in y. So by looking up the *fst* part of r one can decide which x is in *fst* part or *snd* part of y. The extra information r is essentially the realizer of $[x, y] : \mathit{Sub}'$, if Sub' is a class defined by an *extended* CIG inductive definition through the definition equation of Sub deleting the modal operators. See 2.6.2 for further discussion.

(11) Programming on data types: Since a class is an object, we are free to program on data types (classes). For example, we can define a function $\mathit{vector}(n, X)$ which returns the class of lists of X whose length is n by ordinary recursion from Cartesian products such as

$$\mathit{vector}(n, X) = \mathit{cond}(n = 1, X; t, X \times \mathit{vector}(n - 1, X)).$$

An application of programming on data types is implementation of modules. For example, an implementation of a module of stack by lists is given by

$$\begin{aligned} stack(x) &= list(List(x), \Lambda(push), \Lambda(pop), empty), \\ empty &= nil, \\ pop(s) &= list(fst(s), snd(s)), \\ push(a,s) &= pair(a,s). \end{aligned}$$

Note that *push*, *pop*, and *empty* are polymorphic.

(12) Finitely generated CCC: We can define the class of classes generated by given finite numbers of classes and class formation operators. For example the objects of CCC (Cartesian Closed Category) generated from $X_1, \ldots, X_n$ is defined by the CIG definition

$$\begin{aligned} &\textbf{deCIG } x : CCC(X_1, \ldots, X_n) \\ &\equiv equal(x, X_1) \to \top, \\ &\quad \ldots \\ &\quad equal(x, X_n) \to \top, \\ &\quad equal(x, \{\}) \to \top, \\ &\quad t \to \Diamond[\exists y, z.(x = y \times z \wedge y : CCC(X_1, \ldots, X_n) \wedge z : CCC(X_1, \ldots, X_n)) \\ &\qquad\qquad \vee \\ &\qquad\quad \exists y, z.(x = y \to z \wedge y : CCC(X_1, \ldots, X_n) \wedge z : CCC(X_1, \ldots, X_n))]. \end{aligned}$$

The hom set from an object x to y is given by $hom(x,y) = x \to y$, the identity morphism is $id = \Lambda(\lambda(x).x)$, and composition is $f \circ g = \Lambda(\lambda(x).app{*}(f, (app{*}(g,x))))$. So a Cartesian closed category generated from $X_1, \ldots, X_n$ is given by a list $list(CCC(X_1, \ldots, X_n), \Lambda(hom), id, \Lambda(\circ))$. Note that the identity morphism and composition are polymorphic.

2.4.2. Axioms of Join and Product

In almost all cases, CIG definition is enough to implement useful data types as classes. But **PX** has another kind of class formation method. We assume **PX** has a basic function Σ whose arity is two, and the axiom

(Join)

$$\begin{aligned} &\forall a : A.Class(app{*}(f,a)) \supset \\ &\quad Class(\Sigma(A,f)) \wedge \forall x.(x : \Sigma(A,f) \supset\subset \nabla(a\ .\ b) = x.(a : A \wedge b : app{*}(f,a)))). \end{aligned}$$

This corresponds to Martin-Löf's dependent sum. Note that this does not maintain the existence of the union of the classes $app*(f,a)$ such that $a:A$. We *cannot* prove existence of union in **PX**. See 2.6.2 for a discussion.

By the aid of this axiom and CIG, we can derive the existence of dependent products (Beeson 1985). But, for symmetry, we introduce a class constructor Π and the axiom of dependent products:

$(Product)$

$$\begin{aligned}&\forall a:A.Class(app*(f,a)) \supset\\ &\qquad Class(\Pi(A,f)) \wedge \forall x.(x:\Pi(A,f) \supset\subset \forall a:A.app*(x,a):app*(f,a)).\end{aligned}$$

By these two dependent types and the types introduced in the previous section, we can interpret an *intensional* version of Martin-Löf's type theory $\mathbf{ML}_0$ without the type of well-orderings. It is quite easy to define the type of well-ordering by CIG and Join. But, to model extensionality, we must work some more as did Beeson 1985 and Smith 1984.

Another application of $(Join)$ is a class of implementations of data types. This will be examined in 5.1.

2.4.3. Graph and value class: the minimal graph axiom

The axiom system of **PX** does not include any axioms which maintain the fact that a recursively defined function is the minimal fixed point of its definition. It seems that structural induction is sufficient to prove properties of programs rather than fixed point induction, for a function is normally equipped with an intended domain. So we do not include it as an axiom of **PX**. But, in this section, we will show a way of axiomatizing it *partly*. In our formulation, fixed point induction for a recursively defined function is an instance of CIG induction for an inductively defined class that represents the graph of the function. Instead of introducing a new induction principle for expressing the minimality of a recursively defined function, we introduce an axiom maintaining that an inductively defined class constructed from the function is extensionally equal to the graph of the function. This formulation of fixed point induction was inspired by Constable and Mendler 1985.

First we define the value class of an expression and the graph of a function. The *value class* of an expression e is defined by

$$Value(e) = \{y|e=y\}.$$

The *graph* of a function f is defined by

$$Graph(f) = \{a_1,\ldots,a_n,b|f(a_1,\ldots,a_n)=b\}.$$

Let $A,\ B$ be classes, then $A =_{ext} B$ is $\forall x.(x : A \supset\subset x : B)$. We say A and B are extensionally equal, when $A =_{ext} B$ holds. Then the following holds:

$$\begin{aligned}&\mathit{Value}(fn(e_1, \ldots, e_n))\\ &\qquad =_{ext} \{a|\Diamond\exists a_1 : \mathit{Value}(e_1), \ldots, a_n : \mathit{Value}(e_n).[a_1, \ldots, a_n, a] : \mathit{Graph}(f)\},\end{aligned}$$

$$\begin{aligned}&\mathit{Value}(cond(e_1, d_1; \ldots; e_n, d_1)) =_{ext}\\ &\quad \{a|\Diamond[\exists a_1 : \mathit{Value}(e_1).(a_1 : T \wedge a : \mathit{Value}(d_1))\\ &\qquad \vee\\ &\qquad \exists a_1 : \mathit{Value}(e_1), a_2 : \mathit{Value}(e_2).(a_1 = nil \wedge a_2 : T \wedge a : \mathit{Value}(d_2))\\ &\qquad \vee\\ &\qquad\quad \ldots\\ &\qquad \vee\\ &\qquad \exists a_1 : \mathit{Value}(e_1), a_2 : \mathit{Value}(e_2), \ldots, a_n : \mathit{Value}(e_n).\\ &\qquad\qquad (a_1 = nil \wedge a_2 = nil \wedge \ldots \wedge a_n : T \wedge a : \mathit{Value}(d_n))]\}\end{aligned}$$

$$\begin{aligned}&\mathit{Value}(let\ p_1 = e_1, \ldots, p_n = e_1\ in\ e)\\ &\quad =_{ext} \{a|\Diamond\exists a_1 : \mathit{Value}(e_1), \ldots, a_n : \mathit{Value}(e_n).\\ &\qquad\qquad (\nabla p_1 = a_1, \ldots, p_n = a_n.a : \mathit{Value}(e))\},\end{aligned}$$

$$\mathit{Graph}(\lambda(a_1, \ldots, a_n)(e)) =_{ext} \{a_1, \ldots, a_n, y|y : \mathit{Value}(e)\}.$$

By the aid of these equations a value class or a graph of any expressions or function can be decomposed into value classes and graphs of smaller expressions and functions, as far as the outermost construct of the expression is an application, *cond*, *let*, or an abstraction. So a value class of any expressions can be definable by the foregoing class notations from the value classes of Λ-notations and the graphs of basic functions. We do not decompose the value class of a Λ-notation that depends on the implementation of interpreter, although it is possible. Assume that f is a function defined by a recursive definition in DEF_{i+1}:

(A) $$f(a_1, \ldots, a_n) = e.$$

The right hand side may include the function symbol f. So by the above equations, we can construct a nested class notation $C_e(a_1, \ldots, a_n, X_0)$, considering X_0 as the graphs of f, from the graphs of functions in DEF_i and the value class of Λ-notations so that

$$C_e(a_1, \ldots, a_n, \mathit{Graph}(f)) =_{ext} \mathit{Value}(e).$$

Hence, the minimality of *Graph*(f) can be stated by

(MGA) $$\mathit{Graph}(f) =_{ext} \mu X_0\{a_1, \ldots, a_n, x | x : C_e(a_1, \ldots, a_n, X_0)\}.$$

We call this the *Minimal Graph Axiom* (MGA). Note that CIG induction for the class of the right hand side is fixed point induction for the recursive definition of (A). Although (MGA) is not included in the basic axiom system of **PX**, it holds in the standard semantics of **PX**. So it may or may not be included as an axiom of **PX**, as you like. Officially, we do *not* include this as an axiom of **PX**.

To illustrate the above method, we will compute the graph of Morris's function (see Manna 1974, example 5-27):

$$f(x, y) = \mathit{cond}(\mathit{equal}(x, 0), 1; t, f(x - 1, f(x, y))).$$

Let us denote the right hand side of this equation by e and let X_0 be the graph of the function f. Then $C_e(x, y, X_0)$ is denoted by

$$\mathit{Zero}(x) = \{a | \mathit{equal}(x, 0) \to a = t; t \to a = \mathit{nil}\},$$

$$\begin{aligned}
&\mathit{Value}(\mathit{cond}(\mathit{equal}(x, 0), 1; t, f(x - 1, f(x, y)))) \\
&\qquad =_{ext} \{a | \Diamond[\exists a_1 : \mathit{Zero}(x).(a_1 : T \wedge a = 1) \\
&\qquad\qquad \vee \\
&\qquad\qquad \exists a_1 : \mathit{Zero}(x), a_2 : \{t\}. \\
&\qquad\qquad\quad (a_1 = \mathit{nil} \wedge a : \mathit{Value}(f(x - 1, f(x, y))))]\},
\end{aligned}$$

$$\begin{aligned}
&\mathit{Value}(f(x - 1, f(x, y)) \\
&\qquad =_{ext} \{a | \Diamond\exists a_1 : \{x - 1\}, a_2 : \{a_2 | [x, y, a_2] : X_0\}.[a_1, a_2, a] : X_0\}.
\end{aligned}$$

Then by means of the properties of conditional formulas and modal operator, we see

(B) $$\begin{aligned}
&C_e(x, y, X_0) \\
&\quad =_{ext} \{a | \mathit{equal}(x, 0) \to a = 1; \\
&\qquad\qquad t \to \Diamond\exists a_2.([x, y, a_2] : X_0 \wedge [x - 1, a_2, a] : X_0)\}.
\end{aligned}$$

Let $\emptyset$ be the empty class $\{x, y, z | \bot\}$. Then the first approximation of the graph is given by

$$\begin{aligned}
G_0 &= \{x, y, z | z : C_e(x, y, \emptyset)\} \\
&= \{x, y, z | \mathit{equal}(x, 0) \to z = 1; t \to \bot\}.
\end{aligned}$$

Since the class $C_e(x, y, G_0)$ is empty for $x > 0$,

$$G_1 = \{x, y, z | z : C_e(x, y, G_0)\} = G_0.$$

Hence the minimal fixed point of the operation on classes,

$$X \longmapsto \{x, y, z | z : C_e(x, y, X)\},$$

is just G_0. Actually, by the aid of CIG induction for the class of the right hand side of (B), we can prove

$$\mu X_0\{x, y, z | z : C_e(x, y, X_0)\} =_{ext} \{x, y, z | equal(x, 0) \rightarrow z = 1; t \rightarrow \bot\}.$$

Hence by (MGA) and the definition of *Graph*(f), we see that

$$\begin{aligned} equal(x,0) &\rightarrow \forall z.(f(x,y) = z \supset\subset z = 1); \\ t &\rightarrow \forall z.(f(x,y) = z \supset\subset \bot). \end{aligned}$$

Hence

$$equal(x, 0) \rightarrow f(x, y) = 1; t \rightarrow \neg E(f(x, y)).$$

In the usual minimal fixed point semantics of recursive functions, which employs the call-by-name computation rule, Morris's function is the constant function $\lambda x.1$. Since **PX** employs the call-by-value computation rule, the minimal fixed point is as above instead of a constant function.

MGA can determine graphs of functions defined without *app* and *app*∗ but *not* for functions defined with them. Although the graphs of *app* and *app*∗ are *fixed* in MGA, in reality their graphs are defined through definitions of functions. For example, the following function *search*1 is undefined for the singleton list (1) in Lisp, but there is a model of **PX** with MGA in which the graph of *search*1 is the function which searches a leaf labeled by 1 of a tree so its value for the list (1) is t:

$$\begin{aligned} search1(x) = cond(&equal(x, 1), t; \\ &atom(x), nil; \\ &t, or(search1(fst(x)), forall(\Lambda(search1), (list(x, x)))) \end{aligned}$$

$$forall(x, y) = cond(y = nil, nil; t, or(app{*}(x, fst(x)), forall(x, snd(y)))).$$

It is possible to formulate a complete fixed point induction for recursive function definitions. But the above example shows that if *app* or *app*∗ concerns a definition, then fixed point induction must be formulated with a simultaneous recursive definition with *app* and *app*∗. Furthermore, it would need the concept of function environment. As these are complicated, so we do not attempt to formalize them.

2.5. Choice rule

In the implementation of **PX**, we may use another important rule, although it is not counted as a basic inference rule of **PX**. The rule is a version of the extended Church's rule of Troelstra 1973:

$$(choice) \qquad \frac{\Gamma \Rightarrow \exists x_1, ..., x_n.A}{\{\ \} \Rightarrow \exists f.\forall a_1, ..., a_m.(\bigwedge \Gamma \supset \nabla(x_1...x_n) = app{*}(f, a_1, ..., a_m).A)},$$

where formulas of Γ are of rank 0, formulas of Γ and A do not have partial variables, $a_1, ..., a_n$ are all of the free variables in the upper sequent, and $\bigwedge \Gamma$ is the conjunction of the formulas of Γ.

If this is interpreted as a semantically valid rule, then our semantics would contradict classical logic. So we do *not* regard it as a semantically valid rule. Indeed, this does *not* hold in our intended semantics; on the contrary, all of the others hold. This rule should be interpreted as a derived rule, i.e., the set of derivable sequents of **PX** is closed under the rule. This fact will be proved in 3.1.

In the implementation of **PX**, the choice rule is presented as an declaration of a choice function as follows:

$$(choice2) \qquad \frac{\Gamma \Rightarrow \exists x_1, ..., x_n.A}{\Gamma \Rightarrow \nabla(x_1...x_n) = f(a_1, ..., a_m).A},$$

where f is a *new* function of arity n.

2.6. Remarks on the axiom system

In this section, we will give two remarks on the system to help the reader's understanding of the system. One is about the difference between the logic of partial term (LPT) and the logic of partial existence (LPE). Another is about information transparencies of classes and limitations caused by it.

2.6.1. The logic of partial terms versus the logic of partial existence

The logic in Scott 1979 looks essentially the same as our LPT. (Moggi 1988 gives a relationship between them, and extensive discussions and studies on various forms of LPT and LPE.) But it is a logic of partial existence (LPE) and there is a philosophical difference between LPT and LPE. The main difference lies in its semantics. The semantics of LPE supposes the existence of partial elements like $\perp$ of cpo. The semantics of LPT does not assume the existence of partial elements. In LPT, all values are total. LPE has quantifications over partial elements, but

LPT has only quantifications over total elements. In LPE, interpretation of every expression is always defined as a value in its semantic domain. If an expression is "undefined", then its value should be an "undefined value" in its semantic domain. In LPT, an expression may fail to denote a value. So interpretation of terms in LPT is partial. Since the semantic domain of LPT may be ordinary sets, LPT seems closer to the traditional semantics of logic. It is possible to represent partial elements in the frame work of LPT. Plotkin 1985 represented the partial value of an expression e by $\lambda x.e$, where x is a variable not occurring in e. In **PX** we can represent it by $\Lambda(\lambda(x)(e))$. Note that this expression has a value as long as e does not have a partial variable. So a domain of partial values of D is defined by the lifting $\{nil\} \rightharpoonup D$. This domain of partial values is slightly different from the domain of partial values in the sense of Fourman 1977 and Scott 1979. If a predicate $P(x)$ over a domain D has the property $\forall x, y \in D.(P(x) \supset x = y)$, then there is a partial value p which satisfies the property $\forall x \in D.(P(x) \supset x = p)$ in their theories. But this is not true, when we use $\{nil\} \rightharpoonup D$ as the domain of partial values of D.

2.6.2. Transparency of classes

The reason why we restrict CIG templates to rank 0 formulas is that we do not wish that a formula $e : A$ has hidden information. We wish classes to be transparent. For example, if we allow a CIG inductive definition of a class NN of natural numbers like

$$x : NN \equiv_{def} x = 0 \vee \exists y : NN.x = suc(y),$$

then to show $x : NN$, we need a realizer r which realizes $x : NN$, i.e., information by which we can see how x is constructed from 0 by *suc*. (See the next chapter for the exact definition of realizers.) Suppose we extracted a program f from a proof of $\forall x : NN.\exists y.A(x, y)$. Then f needs an extra input r which realizes the fact $x : NN$ besides the input x, i.e. f satisfies only the property

$$\forall x, r.(r \text{ realizes } x : NN \supset \nabla(x\ y) = f(x, r).A(x, y)).$$

For NN and also for almost all inductively defined data types, such extra information r has a structure isomorphic to the input x itself, although not identical. So the extra input r is redundant. Furthermore, for any class C defined by such an extended inductive definition, the class $\{x, r | r \text{ realizes } x : C\}$ can be defined by CIG inductive definition just we did in the example (10) of CIG inductive definition in 2.4.1. Namely, if one wishes to define a class with hidden information, one can do that with a CIG inductive definition which explicitly includes such "hidden information" as an argument of the class. (See Feferman 1979 for a systematic method to eliminate hidden information.)

A drawback of the restriction is that we cannot define a union of classes. The axiom (*Join*) maintains the existence of a disjoint union of a family of classes. If $app{*}(f,x)$ is a class for each $x : A$, an element of the join $\Sigma(A,f)$ is a pair of x and y such that $x : A$ and $y : app{*}(f,x)$ rather than an element of $app{*}(f,x)$ such that $x : A$. The union of the classes $\bigcup x : A.app{*}(f,x)$ is defined by

$$\{y|\exists x : A.pair(x,y) : \Sigma(A,f)\}.$$

But this is not allowed in **PX**. Since we may replace the atomic formula $y : \bigcup x : A.app{*}(f,x)$ by the formula $\exists x : A.pair(x,y) : \Sigma(A,f)$, it is always possible to replace a union by a join.

2.6.3. Total variables versus partial variables

There are different opinions on free variables in LPT and related systems. In the systems of Beeson 1981, 1985, 1986 and Moggi 1986, 1988, variables always have values. On the contrary, free variables of LPT of Plotkin 1985 need not have values. (Free variables of LPE need not have values.) **PX** has partial variables and total variables both. The reason why we introduce both is for practical purposes.

In ordinary logic, in which terms always have values, the axiom $x = x$ and the axiom scheme $e = e$ are equivalent. In a LPT whose free variables are total variables, the axiom scheme $e = e$ is stronger than the axiom $x = x$, since a term e which does not have a value cannot be substituted for x. Partial variables can be a substitute for metavariables to some extent and semantics for partial variables is simpler than semantics for metavariables. Partial variables in **PX** are used to describe axioms in an expandable data base of miscellaneous axioms. Even though users do not extend the data base, they can use partial variables to state theorems about LPT. If only total variables were allowed in **PX**, they could not be able to state a fact

$$E(cond(e_1, d_1; e_2, d_2)) \supset (E(e_1) \wedge E(d_1)) \vee (E(e_1) \wedge E(e_2) \wedge E(d_2))$$

by a single theorem. Considering e_1, e_2, d_1, and d_2 as partial variable, we can state it by a single formula. Partial variables are useful to state properties about program constructs.

The reason why PX has total variables as well is again practical. Theoretically, we do not need free total variables, since a partial variable p turns to a total variable by adding an extra existence assumption $E(p)$ or $p : V$. In practice, partial variables are hardly used in a formula as a specification. Actually, no partial variable is used in the examples in this book. In practice, it is rather messy to handle the extra existence assumptions on partial variables. When total variables are used, we may omit such messy existence assumptions. So total variables are useful to state formulas as specifications.

Since both total variables and partial variables are useful in different purposes, we introduced both. It is not difficult to handle them in practice, since they are syntactically distinguishable.

3 Realizability

There are many ways giving computational meaning to constructive proofs. We use realizability to give computational meaning to proofs of **PX** as in Nepeĭvoda 1982, Hayashi 1983, 1986, Beeson 1985, and Sato 1985. We use a slightly modified **q**-realizability. Our realizability is called the **px**-realizability, and was introduced in Hayashi 1983, 1986. It was referred to there as the **q**-realizability, but in this book we refer to it as **px**-realizability. The advantage of **px**-realizability is that if a program f is extracted from a proof of a theorem $\forall x.\exists y.A$, then f satisfies $\forall x.\nabla y = f(x).A$ in the sense of classical logic (4.1 and appendix A), and rank 0 formulas are realizable iff this holds classically. The contents of this chapter are rather technical, so readers who are not familiar with the subject or not interested in a detailed proof of the soundness result may read the definition of realizability briefly and put off reading the rest of this chapter until the knowledge about the extraction algorithm is required in the next chapter. In the rest of this book, only the extraction algorithm in 3.2 and its soundness result will be needed.

The definition of **px**-realizability in 3.1 is presented only to explain the idea of the soundness theorem. This is *not* the version of **px**-realizability that the actual **PX** uses. The actual **px**-realizability for **PX** is presented in 3.2. Except in 3.1, **px**-realizability means the refined **px**-realizability of 3.2.

3.1. px-realizability

Realizability of constructive mathematics goes back to Kleene's work in the forties. Now quite a few variants are known (Troelstra 1973, Beeson 1985). Furthermore, their intrinsic meanings have been explored by the aid of categorical logic (Hyland 1982, McCarty 1984). We present a realizability called **px**-realizability and prove its soundness for **PX** in this subsection. As remarked above, this is *not* the **px**-realizability and extraction algorithm used in the implementation of **PX**. The actual version makes more distinctions of cases for optimization of extracted programs.

Now we give the **px**-realizability. For each formula A, we assign a formula $a\ \mathbf{x}\ A$, $A^{\mathbf{x}}$ for short. We call it the **px**-realizability or **px**-realization of A. When $a\ \mathbf{x}\ A$ holds, we say "a is a *realizer* of A" or "a *realizes* A". An important point is that a is a total individual variable not occurring in A. So a realizer must be a value (object). The variable a will be called a *realizing variable* of A. A realizing variable may be an arbitrary individual variable that does not occur in the realized formula. But, for convenience, we assume that a *new* variable is assigned to each formula as its own realizing variable. Strictly speaking, we extend

PX by adding new individual variables. The new variables are exclusively used as realizing variables. So in $a \mathbf{x} A$, a should be a new variable, but A must not have any new variables. Furthermore, we assume that if two formulas are α-convertible then their realizing variables are the same.

We write $e \mathbf{x} A$ for $(a \mathbf{x} A)[e/a]$. For simplicity, we will often say simply "realizability" instead of "**px**-realizability". We define realization of formulas by the induction on the complexity of formulas. We will define it so that $FV(a \mathbf{x} A)$ is a subset of $\{a\} \cup FV(A)$.

Definition 1 (px-realizability)

1. A is of rank 0. Then $a \mathbf{x} A$ is $A \wedge a = nil$.
2. $a \mathbf{x} A_1 \wedge \ldots \wedge A_n$ is $\nabla(a_1 \ldots a_n) = a.((a_1 \mathbf{x} A_1) \wedge \ldots \wedge (a_n \mathbf{x} A_n))$.
3. $a \mathbf{x} A \supset B$ is $E(a) \wedge A \supset B \wedge \forall b.((b \mathbf{x} A) \supset \nabla c = app{*}(a,b).c \mathbf{x} B)$.
4. $a \mathbf{x} A_1 \vee \ldots \vee A_n$ is $\nabla(b \ . \ c) = a.Case(b, c \mathbf{x} A_1, \ldots, c \mathbf{x} A_n)$.
5. $a \mathbf{x} \forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.A$ is
$$E(a) \wedge \forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.\nabla y = app{*}(a, a_1, \ldots, a_m).y \mathbf{x} A,$$
where $a_1, \ldots, a_m$ is the concatenation of $FV(\vec{x}_1), \ldots, FV(\vec{x}_n)$.
6. $a \mathbf{x} \exists \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.$ is
$$\nabla(a_1 \ldots a_m \ b) = a.(\vec{x}_1 : e_1 \wedge \ldots \wedge \vec{x}_n : e_n \wedge b \mathbf{x} A),$$
where $a_1, \ldots, a_m$ is the concatenation of $FV(\vec{x}_1), \ldots, FV(\vec{x}_n)$.
7. $a \mathbf{x} \nabla p_1 = e_1, \ldots, p_n = e_n.A$ is $\nabla p_1 = e_1, \ldots, p_n = e_n.a \mathbf{x} A$.
8. $a \mathbf{x} e_1 \to A_1; \ldots; e_n \to A_n$ is $e_1 \to a \mathbf{x} A_1; \ldots; e_n \to a \mathbf{x} A_n$.

The following lemmas will be used in the proof of the soundness theorem. They are easily proved by induction on the complexity of formulas.

Lemma 1. *In* **PX**, *we can prove* $e \mathbf{x} A \Rightarrow E(e)$ *and* $e \mathbf{x} A \Rightarrow A$. *As a consequence of the former,* $e \mathbf{x} A$ *and* $\nabla a = e.a \mathbf{x} A$ *are logically equivalent in* **PX**.

Lemma 2 (substitution lemma for px-realizability). *Let* σ *be a substitution which does not substitute for free variables of* e. *Then* $e \mathbf{x} (A\sigma)$ *and* $(e \mathbf{x} A)\sigma$ *are logically equivalent in* **PX**.

Let e be an *expression*, and let $\Gamma \Rightarrow A$ be a sequent. Then $e \mathbf{x} (\Gamma \Rightarrow A)$ is a *pair* of sequents $\Gamma \Rightarrow A$ and $\Gamma^{\mathbf{x}} \Rightarrow e \mathbf{x} A$, where $\Gamma^{\mathbf{x}}$ is the set $\{A^{\mathbf{x}} | A \in \Gamma\}$. The expression e is called a (provable) *realizer* of the sequent $\Gamma \Rightarrow A$, provided $e \mathbf{x} (\Gamma \Rightarrow A)$ is provable, i.e. both of the two sequents are provable. Then we can prove the following theorem, which gives the mathematical foundation of program extraction with **PX**.

Theorem 1 (Soundness of px-realizability). *If* $\Gamma \Rightarrow A$ *is a provable sequent of* **PX** *without partial variables, then we can find an expression* e *from its proof such that* e **x** $(\Gamma \Rightarrow A)$ *is also provable in* **PX**. *Furthermore, we may assume* $FV(e)$ *is a subset of* $FV(\Gamma^{\mathbf{x}} \cup \{A\})$.

From the soundness of **px**-realizability, we can easily conclude the following corollary.

Corollary. **PX** *is closed under the rule of* (*choice*). *Especially, if*

$$\forall[x_1, \ldots, x_n] : e.\exists y.F(x_1, \ldots, x_n, y)$$

is a provable sentence (closed formula) in **PX**, *then there is a closed function* fn *such that*

$$\forall[x_1, \ldots, x_n] : e.\nabla y = fn(x_1, \ldots, x_n).F(x_1, \ldots, x_n, y)$$

is also provable in **PX**.

Proof. We prove that **PX** is closed under (*choice*). Assume that e_0 provablely realizes the upper sequent of the rule in 2.5. By the condition on free variables of realizers, we may assume $FV(e_0) \subset FV(\Gamma^{\mathbf{x}}) \cup FV(\exists \vec{v}_1, \ldots, \vec{v}_n.A)$ holds. Substitute *nil* for the realizing variable of $\Gamma^{\mathbf{x}}$, and let us denote it by Γ'. Then Γ' is logically equivalent to Γ. Hence, we may assume that $FV(e_0) \subset \{a_1, \ldots, a_m\}$ and $\Gamma^{\mathbf{x}} = \Gamma$. Set

$$f = \Lambda(\lambda(a_1, \ldots, a_m)(let\ (b_1 \ldots b_n\ c) = e_0\ in\ list(b_1, \ldots, b_n)))$$

and

$$d = \Lambda(\lambda(a_1, \ldots, a_m).\Lambda(\lambda()(let\ (b_1 \ldots b_n\ c) = e_0\ in\ c))),$$

where $b_1, \ldots, b_n$ are fresh variables. Let e be $list(f, d)$. Then e is a closed expression and provably realizes the lower sequent. The first claim follows from this from the definition of realizability by lemma 1. The second claim of the corollary holds, for we defined f explicitly. □

The condition on variables in theorem 1 is necessary to validate the **px**-realizability of $(\forall I)$ and $(\supset I)$, etc. The variables that appear in formulas as specifications are normally total variables. Partial variables of **PX** are mainly used as a substitute for metavariables. Namely if a formula with a partial variable is provable, then any formula obtained by substituting any expressions for the variables is also provable. Assume that a partial variable x appears in a specification formula A and e is a realizer of A. When we substitute any expression e' for x, $e[e'/x]$ **x** $A[e'/x]$ should hold, but the variable condition of the theorem prevents this. Plotkin's lifting technique provides a way to avoid this. Let y be a total variable. By substituting an expression $app{*}(y)$ for x of A, A turns out to be a formula without partial variables. Assume e_1 is extracted as a proof of $A[app{*}(y)/x]$. The expression $\Lambda(\lambda()(e'))$ always has a value, provided e' does not have partial variables. So substituting it for y, we see $e_1[\Lambda(\lambda()(e'))/y]$ **x** $A[e'/x]$. Hence $app{*}(y)$ can be a substitute for a metavariable for expression *without* partial variables. If one wishes e' to have a partial variable, then one can use the same technique again.

We can eliminate partial variables in a proof of a provable sequent without partial variables by the aid of the following lemma.

Lemma 3. *If a provable sequent does not have partial variables, then it has a proof in which any partial variables do not appear.*

Proof. If partial variables are replaced by arbitrary constants, then all axioms and rules remain valid except (*inst*). So we eliminate substitutions for partial variables in (*inst*) by substituting γ_i for the variables z_i through the proof for every $i = 1, \ldots, p$. We make this change from the bottom to the top. Finishing this change, we substitute arbitrary constants for the remaining partial variables. Then it turns out to be a proof without any partial variables. □

In the following proof, we assume that all proofs have no partial variables, when the conclusion has no partial variables.

Proof of Theorem 1. The proof is by induction on the complexity of proofs. Note that we have to construct an expression for the lower sequent of each inference rule and prove that it *provably* realizes the sequent. In the rest of the proof, all realizers are provable realizers.

Since all axioms are of rank 0 except (= 5), their realizers are *nil*. A realizer of (= 5) is

$$cond(equal(a,b), pair(1,nil); t, pair(2,nil)).$$

Hence only inference rules remain. If the conclusion of an inference rule is of rank 0, then *nil* is a realizer of the lower sequent. We may assume that the conclusions of inference rules are of rank 0 in the rest of the proof. Hence we do not go over the inference rules, whose conclusions are always of rank 0. The realizers we construct below may or may not satisfy the additional condition on free variables. When one does not satisfy the condition, we substitute appropriate constants for free variables which are not free variables of the sequent. We examine each inference rule and construct its realizer e. In almost all cases, it is routine to prove that the expression we construct is a realizer. We leave details of the proof to the reader, except in some difficult or subtle cases. In particular, we will give a detailed proof of the realization of the axioms and rule of CIG.

$(= 4)$ Take the realizer of the left upper sequent as e. Lemma 2 implies that this is a realizer of the lower sequent.

(assume) Take the realizing variable of A as e.

$(\bot)$ Take nil as e.

$(\mathit{thinning})$ Take the realizer of the upper sequent as e.

(inst) Let e_0 be a realizer the first upper sequent. Let τ be the substitution that substitutes the realizing variables of $A\sigma$ for the realizing variable of for every A of Γ. Set $e = e_0\sigma\tau$.

(cut) Let e_1 and e_2 be realizers of the left upper sequent and the right upper sequent, respectively. Set

$$e = \mathit{let}\ a = e_1\ \mathit{in}\ e_2,$$

where a is the realizing variable of A.

(alpha) Take the realizer of the upper sequent as e.

$(\mathit{replacement})$ Take the realizer of the left upper sequent as e. Note that the formulas B_1, B_2 must be rank 0 formulas. We can prove that e is a realizer by proving that if B and C are rank 0 formulas, then

$$\{\mathit{Env}_{A[*]}[B \supset C], a\ \mathbf{x}\ A[B]_+\} \Rightarrow a\ \mathbf{x}\ A[C]_+,$$

$$\{\mathit{Env}_{A[*]}[B \supset C], a\ \mathbf{x}\ A[C]_-\} \Rightarrow a\ \mathbf{x}\ A[B]_-$$

hold. This can be proved by induction on the complexity of the context $A[*]$.

$(\wedge I)$ Let e_i be a realizer of $\Gamma \Rightarrow A_i$ for each $i = 1, \ldots, n$. Set

$$e = \mathit{list}(e_1, \ldots, e_n).$$

$(\wedge E)$ Let e_1 be a realizer of the upper sequent. Set

$$e = \mathit{let}\ [a_1 \ldots a_n] = e_1\ \mathit{in}\ \mathit{list}(a_{i_1}, \ldots, a_{i_m}).$$

$(\vee I)$ Let e_1 be a realizer of the upper sequent. Take $\mathit{pair}(i, e_1)$ as e.

($\vee E$) Let e_0 be a realizer of the first upper sequent, and let e_i be a realizer of the $i+1$th upper sequent for each $i = 1, \ldots, n$. Then set

$$e = let\ (c\ .\ r) = e_1\ in\ case(c, e_1[r/a_1], \ldots, e_n[r/a_n]),$$

where a_i is the realizing variable of A_i and c, r are fresh variables.

($\supset I$) Let e_1 realize the upper sequent, and let a be the realizing variable of A. Set

$$e = \Lambda(\lambda(a).e_1).$$

This is a legal expression, for e_1 does not have free partial variables. By the induction hypothesis, we see $\Gamma^{\mathbf{x}} \Rightarrow e_1\ \mathbf{x}\ B$ holds. We prove

$$(\Gamma - \{A\})^{\mathbf{x}} \Rightarrow e\ \mathbf{x}\ A \supset B.$$

$E(e)$ holds by (E 5). Let b be the realizing variable of A. Then, $\{\ \} \Rightarrow app{*}(e, a) = (\lambda(a).e_1)(a) = e_1$ holds by (app) and ($beta$). By the induction hypothesis, lemma 1, and ($\nabla\exists I$), we see $\Gamma^{\mathbf{x}} \Rightarrow \nabla c = app{*}(e, a).c\ \mathbf{x}\ B$. Hence we see

$$(\Gamma - \{A\})^{\mathbf{x}} \Rightarrow \forall a.(a\ \mathbf{x}\ A \supset \nabla c = app{*}(e, a).c\ \mathbf{x}\ B).$$

On the other hand, $(\Gamma - \{A\})^{\mathbf{x}} \Rightarrow A \supset B$ holds by lemma 1.

($\supset E$) Let e_1 and e_2 be realizers of the left upper sequent and right upper sequent, respectively. Set

$$e = app{*}(e_1, e_2).$$

($\forall I$) Let e_1 be a realizer of the upper sequent, and let $a_1, \ldots, a_m$ be the finite sequence of variables obtained by concatenating the sequences of variables $FV(\vec{v}_1), \ldots, FV(\vec{v}_n)$. Set

$$e = \Lambda(\lambda(a_1, \ldots, a_m).e_1).$$

Similarly as in the case of ($\supset I$), we can prove e is a realizer.

($\forall E$) Let e_0 be a realizer of the first upper sequent. Set

$$e = app{*}(e_0, e^1_1, \ldots, e^1_{m_1}, \ldots, e^n_1, \ldots, e^n_{m_n}).$$

Use lemma 2 to prove e is a realizer.

($\exists I$) Let e_0 be a realizer of the first upper sequent. Set

$$e = list(e^1_1, \ldots, e^1_{m_1}, \ldots, e^n_1, \ldots, e^n_{m_n}, e_0).$$

Use lemma 2 to prove e is a realizer.

($\exists E$) Let e_1 and e_2 be realizers the left upper sequent and right upper sequent, respectively. Let $a_1, \ldots, a_m$ be the sequence of variables obtained by concatenating the sequences of variables $FV(\vec{v}_1), \ldots, FV(\vec{v}_n)$. Then set

$$e = \mathit{let}\ (a_1 \ldots a_m\ a) = e_1\ \mathit{in}\ e_2,$$

where a is the realizing variable of A.

($\rightarrow \vee I$) Take the realizer of the rightmost upper sequent as e.

($\rightarrow \vee E$) Let e_0 realize the first upper sequent, and let c_i realize the $i+1$th upper sequent for each $i = 1, \ldots, n$. Let a_i be the realizing variable of A_i for each $i = 1, \ldots, n$. Then set

$$e = \mathit{cond}(e_1, \mathit{let}\ a_1 = e_0\ \mathit{in}\ c_1; \ldots; e_n, \mathit{let}\ a_n = e_0\ \mathit{in}\ c_n)$$

and

$$\Delta = \Gamma \cup \Pi_1 - (\{A_1\} \cup S_1) \cup \ldots \cup \Pi_n - (\{A_n\} \cup S_n).$$

We have to prove $\Delta \Rightarrow e\ \mathbf{x}\ C$. By lemma 1 of 2.3.4, this is equivalent to

$$(*) \qquad \Delta \Rightarrow \nabla a_1 = e_0, \ldots, a_n = e_0 . e_1 \rightarrow c_1\ \mathbf{x}\ C; \ldots; e_n \rightarrow c_n\ \mathbf{x}\ C.$$

By the induction hypothesis on the first upper sequent, we see that

$$\Gamma^{\mathbf{x}} \cup \{e_1 = \mathit{nil}, \ldots, e_i = \mathit{nil}, e_{i+1} : T\} \Rightarrow e_0\ \mathbf{x}\ A_{i+1}$$

holds for each $i = 1, \ldots, n$. Hence, by the induction hypothesis on the right upper sequent, we see that

$$\Gamma^{\mathbf{x}} \cup \{e_1 = \mathit{nil}, \ldots, e_i = \mathit{nil}, e_{i+1} : T\} \cup (\Pi_i^{\mathbf{x}} - \{e_0\ \mathbf{x}\ A_{i+1}\}) \Rightarrow c_i[e_0/a_i]\ \mathbf{x}\ C$$

holds for each $i = 1, \ldots, n$. On the other hand, by the first upper sequent, we can derive the sequent $\Gamma \Rightarrow \mathit{Sor}(e_1, \ldots, e_n)$. Hence, by ($\rightarrow \wedge I$) and (*thinning*) we see that

$$\Delta^{\mathbf{x}} \Rightarrow e_1 \rightarrow c_1[e_0/a_1]\ \mathbf{x}\ C; \ldots; e_n \rightarrow c_n[e_0/a_n]\ \mathbf{x}\ C$$

holds. Hence, by ($\nabla\exists I$), we prove $(*)$.

($\rightarrow \wedge I$) Let d_i be a realizer of the ith upper sequent. Set

$$e = \mathit{cond}(e_1, d_1; \ldots, ; e_n, d_n).$$

$(\to \wedge E)$ Take the realizer of the first upper sequent as e.

$(\nabla\exists I)$ Take the realizer of the first upper sequent as e. Use lemma 2 to prove it is a realizer.

$(\nabla\exists E)$ Let d_1 and d_2 be realizers of the left upper sequent and right upper sequent, respectively. Set

$$e = let\ p_1 = e_1, \ldots, p_n = e_n, a = d_1\ in\ d_2,$$

where a is the realizing variable of A.

$(\nabla\forall I)$ Let d be the realizer of the first upper sequent. Set

$$e = let\ p_1 = e_1, \ldots, p_n = e_n\ in\ d.$$

$(\nabla\forall E)$ Take the realizer of the leftmost upper sequent as e.

Now we give a realization of $(CIG\ ind)$. First, we define an expression $pred(A; f; \vec{b})$, for each CIG template A, function name f, and finite sequence of variables $\vec{b}$. We will simply write $pred(A)$ or $pred(A; f)$ instead of $pred(A; f; \vec{b})$, whenever f,$\vec{b}$ are clear from the context. We call $pred(A)$ the *CIG predecessor* of A.

(1) A is a rank 0 formula without any occurrences of X_0. Then $pred(A)$ is nil,
(2) $pred([e_1, \ldots, e_n] : X_0; f; \vec{b}) = f(e_1, \ldots, e_n, \vec{b})$,
(3) $pred(A \supset B) = \Lambda(\lambda(x).pred(B))$, where x is a new individual variable.
(4) $pred(A_1 \wedge \ldots \wedge A_n) = list(pred(A_1), \ldots, pred(A_n))$,
(5) $pred(\forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A) = \Lambda(\lambda(\vec{x}).pred(A))$, where $\vec{x}$ is the concatenation of the sequences of variables $\vec{v}_1, \ldots, \vec{v}_n$,
(6) $pred(e_1 \to A_1; \ldots; e_n \to A_n) = cond(e_1, pred(A_1); \ldots; e_n, pred(A_n))$,
(7) $pred(\nabla p_1 = e_1, \ldots, p_n = e_n.A) = let\ p_1 = e_1, \ldots, p_n = e_n\ in\ pred(A)$.

Assume e_0 realizes the upper sequent of $(CIG\ ind)$. Define a function f_0 by

$$f_0(a_1, \ldots, a_n, b_1, \ldots, b_m) = let\ r = pred(A; f_0; b_1, \ldots, b_m)\ in\ e_0,$$

where r is the realizing variable of $A[F(\vec{a})/X_0]$ and $\vec{b} = b_1, \ldots, b_n$ is the sequence of all free variables of e_0 except r and $a_1, \ldots, a_n$. Set

$$e = f_0(a_1, \ldots, a_n, b_1, \ldots, b_m).$$

Then e realizes the lower sequent. To prove this, we need the following lemma.

Lemma 4. *Let $F^*(\vec{a})$ be the formula $f_0(\vec{a},\vec{b})$ $\mathbf{x}$ $F(\vec{a})$. Then the following is provable in* **PX***:*

$$\text{(A)} \qquad A[F^*(\vec{a})/X_0] \Rightarrow \mathit{pred}(A; f_0; \vec{b})\ \mathbf{x}\ A[F(\vec{a})/X_0].$$

We will prove this lemma later. Here we use the lemma to prove that e realizes the lower sequent of $(CIG\ ind)$.

$(CIG\ ind)$ Assume the following is provable:

$$(*) \qquad \begin{aligned}&(\Gamma - \{A[F(\vec{a})/X_0], A[\mu X_0\{\vec{a}|A\}/X_0]\})^{\mathbf{x}}\\ &\qquad \cup \{A[F^*(\vec{a})/X_0], A[\mu X_0\{\vec{a}|A\}/X_0]\} \Rightarrow F^*(\vec{a}).\end{aligned}$$

Then, by applying $(CIG\ ind)$ to this sequent, we can derive

$$\{[\vec{a}] : \mu X_0\{\vec{a}|A\}\} \cup (\Gamma - \{A[F(\vec{a})/X_0], A[\mu X_0\{\vec{a}|A\}/X_0]\})^{\mathbf{x}} \Rightarrow F^*(\vec{a}).$$

This means that e realizes the lower sequent of $(CIG\ ind)$. Hence it is sufficient to derive $(*)$. By the induction hypothesis, $\Gamma^{\mathbf{x}} \Rightarrow e_0\ \mathbf{x}\ F(\vec{a})$ is provable. By lemma 4, we can prove

$$\Gamma_0 \cup \{r = \mathit{pred}(A; f_0; \vec{b}), A[F^*(\vec{a})/X_0]\} \Rightarrow e_0\ \mathbf{x}\ F(\vec{a}),$$

where $\Gamma_0 = \Gamma^{\mathbf{x}} - \{r\ \mathbf{x}\ A[F(\vec{a})/X_0]\}$ and r is the realizing variable of $A[F(\vec{a})/X_0]$.

Since $\{A[F^*(\vec{a})/X_0]\} \Rightarrow E(\mathit{pred}(A; f_0; \vec{b}))$ is provable by lemmas 1 and 4, we can prove the following by $(\nabla\forall I)$ and lemma 1 of 2.3.4:

$$(1) \qquad \Gamma_0 \cup \{A[F^*(\vec{a})/X_0]\} \Rightarrow (\mathit{let}\ r = \mathit{pred}(A; f_0; \vec{b})\ \mathit{in}\ e_0)\ \mathbf{x}\ F(\vec{a}).$$

Since $A[\mu X_0\{\vec{a}|A\}/X_0]$ is a rank 0 formula, we see

$$(2) \qquad A[\mu X_0\{\vec{a}|A\}/X_0] \supset (A[\mu X_0\{\vec{a}|A\}/X_0])^{\mathbf{x}}.$$

The sequent $(*)$ follows from (1) and (2).

This is just the sequent $(*)$ by definition of the function f_0. This ends the proof of validity of the realization of $(CIG\ ind)$. □

Now we prove lemma 4. We will prove it by induction for rank 0 formulas.

Proof of Lemma 4

(1) Assume that A is a rank 0 formula without any occurrences of X_0. Then $A[F^*(\vec{a})/X_0]$, and $A[F(\vec{a})/X_0]$ are the same rank 0 formula. So (A) is provable.

(2) Assume A is $[e_1, \ldots, e_n] : X_0$. Then (A) is tautological by the definition of $\mathit{pred}(A; f_0; \vec{b})$.

(3) Assume A is $A_1 \supset A_2$. Let e_0 be $\mathit{pred}(A_2; f_0; \vec{b})$. Since X_0 appears negatively in A_1, $\{A_1[F(\vec{a})/X_0]\} \Rightarrow A_1[F^*(\vec{a})/X_0]$ is provable. Hence, by the induction hypothesis for A_2, we see that $\{A[F^*(\vec{a})/X_0], A_1[F(\vec{a})/X_0]\} \Rightarrow e_0 \mathbf{x} A_2[F(\vec{a})/X_0]$ holds. Since X_0 appears positively in A, $A[F^*(\vec{a})/X_0] \supset A[F(\vec{a})/X_0]$ holds. So we can prove (A).

(4) Assume A is $A_1 \wedge \ldots \wedge A_n$. This case is obvious.

(5) Assume A is $\forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.A$. This case is proved as the case (3).

(6) Assume A is $e_1 \to A_1; \ldots; e_n \to A_n$. We see that

$$\{A[F^*(\vec{a})/X_0]\} \Rightarrow \mathit{Sor}(e_1, \ldots, e_n)$$

holds. Hence, by the induction hypothesis, we see that

$$\{A[F^*(\vec{a})/X_0]\} \Rightarrow e_1 \to \mathit{pred}(A_1) \mathbf{x} A_1; \ldots; e_n \to \mathit{pred}(A_n) \mathbf{x} A_n$$

holds. Hence, by lemma 1 of 2.3.4, (A) is provable.

(7) Assume A is $\nabla p_1 = e_1, \ldots, p_n = e_n.A$. This case is proved as the case (6). □

3.2. A refined px-realizability

The **px**-realizability of the previous section is mathematically simple, but the mathematical simplicity often causes redundancies in practice. We see that

$$\{a | a \mathbf{x} \exists x.A\} = \{(x\ nil) | A\} \cong \{x | A\} \times \{nil\},$$

provided A is a rank 0 formula. The part $\{nil\}$ is redundant. So we wish to define

$$\{a | a \mathbf{x} \exists x.A\} = \{x | A\}.$$

There is large room for similar refinements to the **px**-realizability given in the previous section. We will give a refined **px**-realizability below, which is actually adopted in the implementation described in chapter 7. *In the rest of the book,* **px***-realizability means this refined* **px***-realizability.*

First, to each formula A, we define a nonnegative integer, called the rank of A. We will write it $\mathit{rank}(A)$.

Definition 1 (rank of formula)

1. The rank of atomic formulas and formulas of the forms $\neg A$ and $\Diamond A$ is zero.
2. A is $A_1 \wedge \ldots \wedge A_n$. Set

$$rank(A) = \#\{i | rank(A_i) > 0\},$$

where $\#$ means the cardinality of a set. If all of $A_1, \ldots, A_n$ are of rank 0, then so is A.
3. A is $A_1 \vee \ldots \vee A_n$. If all of $A_1, \ldots, A_n$ are of rank 0, then $rank(A)$ is 1. Otherwise, $rank(A)$ is 2.
4. A is $B \supset C$. If C is of rank 0 then so is A. Otherwise, $rank(A)$ is 1.
5. A is $\forall \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.B$. If B is of rank 0, then so is A. Otherwise, $rank(A)$ is 1.
6. A is $\exists \vec{v}_1 : e_1, \ldots, \vec{v}_n : e_n.B$. Then $rank(A)$ is $m + rank(B)$, where m is the number of variables in $\vec{v}_1, \ldots, \vec{v}_n$.
7. A is $e_1 \to A_1; \ldots; e_n \to A_n$. If all of $A_1, \ldots, A_n$ are of rank 0, then so is A. Otherwise, $rank(A)$ is 1.
8. A is $\nabla p_1 = e_1, \ldots, p_n = e_n.B$. Then $rank(A)$ is $rank(B)$.

It is easy to see that $rank(A)$ is 0 iff A is a rank 0 formula in the sense of 2.2.

Now we define refined **px**-realizability.

Definition 2 (refined px-realizability)

1. Suppose A is of rank 0. Then $a \mathbf{x} A$ is $A \wedge a = nil$.
2. Suppose $A = A_1 \wedge \ldots \wedge A_n$. If $rank(A) = 0$, then let A_i^* be the formula A_i, otherwise, let A_i^* be $a_i \mathbf{x} A_i$. Let $i_1 < \cdots < i_m$ be the sequence of the indices i such that $rank(A_i) \neq 0$. Then $a \mathbf{x} A$ is

$$\nabla[a_{i_1}, \ldots, a_{i_m}] = a.(A_1^* \wedge \ldots \wedge A_n^*).$$

3. $a \mathbf{x} A \supset B$ is

$$E(a) \wedge A \supset B \wedge \forall b.(b \mathbf{x} A \supset \nabla c = fn(a, b).c \mathbf{x} B),$$

where if $rank(A) = 1$ then fn is $app*$ else $rank(A) = 1$ is app.
4. Suppose A is $A_1 \vee \ldots \vee A_n$. If $rank(A_1) = \cdots = rank(A_n) = 0$, then

$$a \mathbf{x} A \equiv Case(a, A_1, \ldots, A_n),$$

otherwise

$$a \mathbf{x} A \equiv \nabla(b\ c) = a.Case(b, c \mathbf{x} A_1, \ldots, c \mathbf{x} A_n).$$

5. $a \mathbf{x} \forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.A$ is

$$E(a) \wedge \forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.\nabla y = app*(a, a_1, \ldots, a_m).y \mathbf{x} A,$$

where $a_1, \dots, a_m$ is the concatenation of $FV(\vec{x}_1), \dots, FV(\vec{x}_n)$.

6. Suppose A is $\exists \vec{x}_1 : e_1, \dots, \vec{x}_n : e_n.A_0$. Let $a_1, \dots, a_m$ be the concatenation of the variables $FV(\vec{x}_1), \dots, FV(\vec{x}_n)$. If $rank(A_0) = 0$, then
$$a \mathbf{\,x\,} A \equiv \nabla[a_1, \dots, a_m] = a.(\vec{x}_1 : e_1 \wedge \dots \wedge \vec{x}_n : e_n \wedge A_0),$$
otherwise
$$a \mathbf{\,x\,} A \equiv \nabla p = a.(\vec{x}_1 : e_1 \wedge \dots \wedge \vec{x}_n : e_n \wedge b \mathbf{\,x\,} A_0),$$
where p is $(a_1 \dots a_m\ b)$ if $rank(A_0) = 1$, and $(a_1 \dots a_m\ .\ b)$ otherwise. The last dot of the latter pattern is an actual dot.
7. $a \mathbf{\,x\,} \nabla p_1 = e_1, \dots, p_n = e_n.A$ is $\nabla p_1 = e_1, \dots p_n = e_n.a \mathbf{\,x\,} A$.
8. $a \mathbf{\,x\,} e_1 \to A_1; \dots ; e_n \to A_n$ is $e_1 \to a \mathbf{\,x\,} A_1; \dots ; e_n \to a \mathbf{\,x\,} A_n$.

Definition 3 (refined CIG predecessor). The definition of CIG predecessors for refined realizability is almost the same as the definition of original CIG predecessors. Only the CIG predecessors for implication and conjunction are changed, as follows:

(3) $pred(A \supset B) = \Lambda(\lambda(x_1, \dots, x_n).pred(B))$, where $x_1, \dots, x_n$ are new individual variables and n is $rank(A)$.
(4) $pred(A_1 \wedge \dots \wedge A_n) = list(pred(A_{i_1}), \dots, pred(A_{i_m}))$, where $i_1, \dots, i_m$ are the same as in clause 2 of definition 2.

We present a detailed description of the actual extraction algorithm below. We do not give its correctness proof, since it is essentially the same as the proof of theorem 1 given above. It is different from the algorithm presented in the proof of theorem 1 in the following respects:

(A) We realize $(CIG\ ind^*)$ instead of $(CIG\ ind)$.
(B) We attach a *tuple with* to each assumption as its realizing variable rather than a single variable. Namely the tuple is the realizing variable, if it is considered as a tuple variable. Recall that when the rank of an assumption is one, then it is a single variable, otherwise it is a list of variables.
(C) The extracted codes are optimized by some partial evaluations.

These are mainly for optimizations of the extracted programs, and the soundness proof of the extracted programs is applicable to it without essential changes.

We have to introduce some auxiliary functions to describe the extractor.

Auxiliary functions

- If A is a formula, then $rvars(A)$ is a sequence of mutually distinct variables of length $rank(A)$. It is considered as a tuple variable that realizes A. We call it the realizing variables of A. We assume $rvars(A)$ and $rvars(B)$ are identical, iff A and B are α-convertible.

- $tuple[e_1, \ldots, e_n]$ is another notation for the tuple $[e_1, \ldots, e_n]$.
- $rpattern(A)$ is the pattern $[a_1, \ldots, a_n]$, where $a_1, \ldots, a_n = rvars(A)$.
- $rpatterns(A_1, \ldots, A_n)$ is the sequence $rpattern(A_1), \ldots, rpattern(A_n)$.
- $newfunc()$ is a new function name.
- $new(n)$ is a sequence of n new total variables.
- $dummy(n)$ is a list of n atoms.
- $upseqs(P)$ is the upper sequents of a proof P.
- $rule(P)$ is the last rule of the proof P.
- $con(P)$ is the conclusion of the proof P.
- $asp(P)$ is the assumptions of the proof P.
- When $rule(P)$ is $(\supset I)$, $disch_{\supset I}(P)$ is the discharged formula of $(\supset I)$.
- When $rule(P)$ is $(inst)$, $subst(P)$ is the substitution σ of the rule of $(inst)$ in 2.3.3.
- $delete0(P_1, \ldots, P_n)$ is the subsequence of $P_1, \ldots, P_n$ obtained by deleting proofs whose conclusions are of ranks 0.
- If $rule(P)$ is $(\wedge E)$, then $indexes_{\wedge E}(P)$ stands for the indexes of the sequence of indexed formulas obtained by deleting all of rank 0 formulas from the sequence of indexed formulas $A_{i_1}, \ldots, A_{i_m}$ of $(\wedge E)$.
- If $rule(P)$ is $(\vee I)$, then $index_{\vee I}(P)$ is the index i in the rule of $(\vee I)$ in 2.3.3.
- $eigenvars_{\forall I}$, etc., denote the sequence of eigenvariables of the rules $(\forall I)$, etc.
- Let $rule(P)$ be $(\forall E)$ and $[e_1/a_1, \ldots, e_m/a_m]$ be the substitution σ of $(\forall E)$ in 2.3.3. Then $instances_{\forall E}(P)$ are the expressions $e_1, \ldots, e_m$. $instances_{\exists I}$ is defined similarly.
- $body_{\exists}(A)$ is the immediate subformula of an existential formula A. $body_{\forall}(A)$, $bodys_{\vee}(A)$, and $bodys_{\rightarrow}(A)$ are similarly defined.
- $conditions(A)$ is the sequence of conditions of a conditional formula.
- Let ϕ be $\nabla p_1 = e_1, \ldots, p_n = e_n.A$. Then $patterns(\phi)$ is $p_1, \ldots, p_n$ and $bindings(\phi)$ is $e_1, \ldots, e_n$.

We construct a realizer of the generalized CIG induction rule $(CIG\ ind^*)$ instead of the plain CIG induction rule $(CIG\ ind)$, but we consider only the following non-simultaneous inductive definition for simplicity. The rule $(CIG\ ind^*)$ is realized in the same way by virtue of simultaneous recursion.

$$(CIG\ dec_1) \qquad \begin{array}{rcl} \mathbf{deCIG}\ \vec{x} : C \equiv_D e_1 & \rightarrow & \phi_{1,1}, \ldots, \phi_{1,q_1}, \\ & \ldots & \\ e_n & \rightarrow & \phi_{n,1}, \ldots, \phi_{n,q_n}. \end{array}$$

$$(CIG\ ind_1^*) \qquad \frac{\Gamma_1 \Rightarrow F, \ldots, \Gamma_n \Rightarrow F}{\bigcup_{i=1}^{n} \tilde{\Gamma}_i \Rightarrow \vec{x} : C \supset F},$$

where $\tilde{\Gamma}_i$ is the set

$$\begin{aligned}\Gamma_i - (\{\vec{x} : D\} & \\ & \cup \{e_1 = nil, \ldots, e_i = nil, e_i : T\} \\ & \cup \{\phi_{i,1}[F/C], \ldots, \phi_{i,q_i}[F/C]\} \\ & \cup \{\phi_{i,1}, \ldots, \phi_{i,q_i}\}).\end{aligned}$$

For a proof P such that $rule(P)$ is $(CIG\ ind_1^*)$, we introduce the following auxiliary functions:

- $template(P)$ is $(e_1; \phi_{1,1}, \ldots, \phi_{1,q_1}), \ldots, (e_n; \phi_{n,1}, \ldots, \phi_{n,q_n})$.
- $pred(A_1, \ldots, A_n)$ is $pred(A_1), \ldots, pred(A_n)$.
- $disch_{CIG}(P_i)$ is the discharged formulas $\phi_{i,1}[F/C], \ldots, \phi_{i,q_i}[F/C]$ of the proof of $\Gamma_i \Rightarrow F$.

Besides these, we use three functions for optimizations

$$Optimize_{case},\ Optimize_{\beta},\ Optimize_{\eta}$$

and a construct **subst** which does substitution and/or builds a *let*-expression in an optimized way. $Optimize_{\beta}$, $Optimize_{\eta}$ do β-reduction and η-reduction, respectively. $Optimize_{case}$ does an optimization such as

$$case(cond(e, 1; t, 2), e_1, e_2) \quad \longmapsto \quad case(e, e_1, e_2).$$

What **subst** does is rather complicated, but essentially it is a *let* form. The value of the expression **subst** $p \leftarrow e$ **in** e_1 is always equivalent to the value of *let* $p = e$ *in* e_1 as far as the both have values. However, if the pattern p (exactly $exp(p)$) occurs only once in e_1, then **subst** does a partial evaluation by replacing p by e, e.g., **subst** $(a\ .\ b) \leftarrow e$ **in** $pair(a, b)$ is just the expression e. Furthermore, if p is the empty pattern () (or nil), **subst** neglects the substitution for p, e.g., **subst** $() \leftarrow e_1$ **in** e_2 is just e_2. To understand results in this book, it suffices to assume **subst** does only these optimizations. See appendix C for a full description.

In the following description of the extraction algorithm, the above auxiliary functions, which are *metalevel* functions in the sense that they handle syntactic entities of **PX**, will be printed in `this typewriter-like font`. Metavariables for such entities are also typed in the same font. On the other hand, the program constructs of **PX** are typed in the font of mathematical formulas. Some metalevel program constructs, like **def, let, case**, are in boldface letters.

The extraction algorithm

```
def extr(P)=
let rank=rank(con(P)), $P_1,\ldots,P_n$=upseqs(P) in
 if rank=0 then $nil$ else
  case rule(P) of
   $(=5)$:   $cond(equal(a,b),1;t,2)$  ; $(=5)$ is the axiom $a=b \vee \neg a=b$
   $(assume)$:  tuple[rvars(con(P))]
   $(\bot)$:  dummy(rank(con(P)))
   $(inst)$:  let $A_1,\ldots,A_n$=asp($P_1$), $\sigma$=subst(P) in
          extr($P_1$)$\sigma$[rvars($A_1\sigma$)/rvars($A_1$),...,rvars($A_n\sigma$)/rvars($A_n$)]
   $(cut)$:  subst  $rpattern$(con($P_1$))$\leftarrow$extr($P_1$) in extr($P_2$)
   $(\wedge I)$:  let $Q_1,\ldots,Q_m$=delete0($P_1,\ldots,P_n$) in
              tuple[extr($Q_1$),...,extr($Q_m$)]
   $(\wedge E)$:  let $a_1,\ldots,a_n$=new(n), $j_1,\ldots,j_p$=indexes$_{\wedge E}$(P) in
              subst  [$a_1,\ldots,a_n$]$\leftarrow$extr($P_1$) in tuple[$a_{j_1},\ldots,a_{j_p}$]
   $(\vee I)$:  if rank(con(P))=1 then index$_{\vee I}$(P)
                    else let i=index$_{\vee I}$(P) in $list$(i,extr($P_i$))
   $(\vee E)$:  if rank(con($P_1$))=1 then
              Optimize$_{case}$($case$(extr($P_1$),extr($P_2$),...,extr($P_n$)))
           else
             let c,r=new(2), $p_1,\ldots,p_{n-1}$=rpatterns(bodys$_\vee$(con(P)))
                in
              Optimize$_{case}$(
                subst  (c r)$\leftarrow$extr($P_1$) in
                    $case$(c,subst  $p_1\leftarrow$r in extr($P_2$)
                                      ,...,
                           subst  $p_{n-1}\leftarrow$r in extr($P_n$)))
   $(\supset I)$:  let $a_1,\ldots,a_m$=rvars(disch$_{\supset I}$(P)) in
              Optimize$_\eta$($\Lambda(\lambda(a_1,\ldots,a_m)$.extr($P_1$)))
   $(\supset E)$:  let f = if rank(con($P_2$))=1 then $app*$ else $app$ in
              Optimize$_\beta$(f(extr($P_1$),extr($P_2$)))
   $(\forall I)$:  let $a_1,\ldots,a_m$=eigenvars$_{\forall I}$(P) in
              Optimize$_\eta$($\Lambda(\lambda(a_1,\ldots,a_m)$.extr($P_1$)))
   $(\forall E)$:  let $e_1,\ldots,e_m$=instances$_{\forall E}$(P) in
              Optimize$_\beta$($app*$(extr($P_1$),$e_1,\ldots,e_m$))
   $(\exists I)$:  let $e_1,\ldots,e_m$=instances$_{\exists I}$(P),u=rank(body$_\exists$(con($P_1$))) in
           if u=0 then tuple[$e_1,\ldots,e_m$]
              else if u=1 then tuple[$e_1,\ldots,e_m$,extr($P_1$)]
                 else  $pair$($e_1$,$pair$($e_2,\cdots,pair$($e_m$,extr($P_1$))$\cdots$))
```

```
(∃E):  let a_1,...,a_m=eigenvars_∃E(P),
           b_1,...,b_u=rvars(body(con(P_1))) in
       if u=0 then subst  [a_1,...,a_m]←extr(P_1) in extr(P_2)
          else if u=1
             then subst  (a_1...a_m b_1)←extr(P_1) in extr(P_2)
                else let b=new(1) in
                   subst  (a_1...a_m . b)←extr(P_1) in
                      subst  (b_1...b_u)←b in extr(P_2)
(→ ∨E):  let c_1,...,c_{n-1}=conditions(con(P_1)),
             p_1,...,p_{n-1}=rpatterns(bodys_→(con(P_1))),
             e=extr(P_1),e_1=extr(P_2),...,e_{n-1}=extr(P_n), in
    cond(c_1,subst p_1←e in e_1;...;c_{n-1},subst p_{n-1}←e in e_{n-1})
(→ ∧I):  let c_1,...,c_{n-1}=conditions(con(P_n)) in
             cond(c_1,extr(P_1);...;c_{n-1},extr(P_{n-1}))
(∇∀I):  let p_1,...,p_m=patterns(con(P_1)),
            e_1,...,e_m=bindings(con(P_1)) in
         let p_1=e_1,...,p_m=e_m in extr(P_1)
(∇∃E):  let p=rpattern(body_∇(con(P_1))),
            p_1,...,p_m=patterns(con(P_1)),
            e_1,...,e_m=bindings(con(P_1)) in
         subst  p_1=e_1,...,p_n=e_m,p←extr(P_1) in extr(P_2)
(CIG ind_1^*):  let a⃗=eigenvars_CIG(P),
                    f=newfunc(),
                    (e_1;A⃗_1),...,(e_n;A⃗_n)=template(P),
                    pred⃗_1=pred(A⃗_1),
                           ...
                    pred⃗_n=pred(A⃗_n),
                    r⃗_1=rpatterns(disch_CIG(P_1)),
                           ...
                    r⃗_n=rpatterns(disch_CIG(P_n)) in
                begin
                   def f(a⃗,b⃗)=cond(e_1,subst  r⃗_1←pred⃗_1 in extr(P_1);
                                                  ...
                                    e_n,subst  r⃗_n←pred⃗_n in extr(P_n))
                end
                f(a⃗,b⃗)
default:  extr(P_1)
end.
```

Besides the above, the extraction algorithm realizes the rule of (*choice*2) of 2.5. As described in 7.2, when one constructs a proof of an existential theorem, say $\exists y : C.A$, with assumptions, say Γ, which are of rank 0, then one can declare a function, say f, for which $\Gamma \Rightarrow \nabla y = f(a_1, \ldots, a_n).(y : C \wedge A)$ is an axiom, where $a_1, \ldots, a_n$ are free variables of Γ and $\exists y.A$. Then the extraction algorithm computes a realizer of the proof of the existential theorem, say `e`, and associates a realizer $\mathtt{e}_1$ with the axiom and attaches an expression $\mathtt{e}_2$ as its definition as follows: if A is of rank 0, then $\mathtt{e}_1$ is *nil* and $\mathtt{e}_2$ is `e`, else $\mathtt{e}_1$ is **subst** *pp* $\leftarrow$`e in b` and $\mathtt{e}_2$ is **subst** *pp* $\leftarrow$`e in y`. where *pp* is `(y b)`, if $rank(A) = 1$, and is `(y . b)` otherwise. We explained only the case when the number of the bound variables of the existential quantifier is one. The general case is treated similarly.

4 Writing programs via proofs

In this chapter, we will give detailed accounts of our methodologies of program extraction with **PX**. We will use petty examples to illustrate our methodologies. A relatively big example will be presented in appendix B, in which we will extract a tautology checker of propositional logic from a proof of its completeness theorem.

In 4.1, we will give an overview how a program is extracted from a proof of **PX**. More precise explanation will be found in chapter 7. In 4.2, we will give a detailed account of how the formulas of **PX** represent "types" through **px**-realizability. In 4.3, we will present a class of recursion called CIG recursion which is the recursion extracted from CIG induction. By the examples of 4.3 and theoretical considerations in 4.4-4.6, the power of the CIG recursion method will be illustrated. Actually, in 4.4, we will show that **PX** can extract *all* partial recursive functions from its proofs, and, in 4.6, we will show that the Hoare logic of total correctness of regular programs is subsumed in **PX**.

4.1. How to extract programs with PX

The first thing one does to extract a program with **PX** is to write a specification by way of a sequent such as

$$\text{(A)} \qquad \Gamma \Rightarrow \exists y : C.A(x, y),$$

where Γ represents the condition on input x, C describes the condition on output y and A describes the expected relation on input and output. The variables of the sequent must be total variables. Writing a constructive proof of (A) and applying the extraction algorithm of 3.2, one gets an expression e of **PX** such that

$$\text{(B)} \qquad \Gamma^{\mathbf{x}} \Rightarrow \nabla y = e.(y : C \land A(x, y))$$

is provable. Namely, e terminates for any input that reflects the real meaning of Γ, i.e., $\Gamma^{\mathbf{x}}$, and the value of e meets the conditions on output. This is also certified by the soundness result in the previous chapter.

Let us explain how features of **PX** serve in the above process. To write a precise specification, one has to decide how data are represented by the data structure of **PX**. S-expressions are known as a universal space of data representation, and CIG inductive definition provides a way of defining recursive data types. The recursive data types defined by CIG inductive definition are much richer than conventional data types, since one can use conditions written by logical formulas with quantifiers. Furthermore, CIG even supports higher order data types as we saw in 2.4.

The inference rules of **PX** represent sufficiently many programming constructs. CIG induction rules represent quite a few recursion schemas and LPT gives a convenient way of writing specifications by formulas, and also serves program constructs as we will see in the rest of this chapter. It is often unnecessary to prove lemmas on properties about data and termination of the algorithm by constructive logic. In **PX** such things may be proved by resorting to classical logic by means of proposition 3 of 2.3.5. This means even that if a proof is incomplete, we can extract a program from it and may complete the proof later by classical logic. Furthermore, we may have the lemmas and termination statement proved by another proof checker. Actually we take advantage of proving them with the proof checker EKL. This will be explained in the rest of this chapter and in chapter 7.

After giving a proof of the specification, we apply the extraction algorithm to it. Then the extracted program meets the specification in the sense that the sequent (B) is also provable in **PX**. Furthermore, it holds in the sense of the semantics given in chapter 6. The semantics is given in the framework of the usual classical logic, so the extracted program meets the specification also in the sense of the usual logic. If Γ is comprised of rank 0 formulas, then $\Gamma^{\mathbf{x}}$ is equivalent to Γ, since the extraction algorithm for the refined realizability does not create realizing variables for rank 0 formulas. In almost all cases in practice, formulas of Γ are of rank 0.

4.2. Curry-Howard isomorphism in PX

The Curry-Howard isomorphism is a paradigm relating constructive logic and programming. It is regarded as a logical foundation of rigid functional programming and type theory in programming languages. (Martin-Löf 1982, Coquand and Huet 1986). Roughly, it says that programming activity corresponds to mathematical activity in the following way:

Constructive logic	Programming and verification
Proof	Verified program
Formula	Specification or Type
Proving theorem	Programming with verification

So the paradigm is also called "proofs as programs", "formulas as types", or "propositions as types". We will see below how they are achieved in **PX**. Concise accounts for the notion can be found in Bates and Constable 1985, Mitchell and Plotkin 1985.

A formula of **PX** represents a "type" through the **px**-realizability. When $e \mathbf{x} A$ holds, we may read it as "e belongs to a type represented by the formula A". When e is extracted from a proof P of A, then we read it as "P represents a program e which has the type represented by A". In short, "P is a program of a type A". Logical connectives and quantifiers that construct formulas correspond to type constructors, and inference rules that construct proofs correspond to program constructs. When a formula is read as a type and a proof is read as a proof by the ordinary realizability, then the correspondence between them is ordinary one as in Martin-Löf 1982. Since we adopted **px**-realizability instead of the usual realizability, our "formulas as types" notion differs from the ordinary one, although the "proofs as programs" notion is almost the same as the ordinary one. We will compare sets of realizers by **px**-realizability with types of typed programming languages below. We use Reynolds 1985 as a standard source of concepts and notations of type disciplines in programming languages.

Each formula A represents a "type" $\{a|a \mathbf{x} A\}$. For simplicity we denote the *set* of realizer of A by $\mathbf{x}(A)$. (Note that $\mathbf{x}(A)$ need not be a class. So $\{a|a \mathbf{x} A\}$ is not our class notation but the usual set notation.)

The type of rank 0 formula A is an extended propositional equality in the sense of Martin-Löf 1982, i.e., it has exactly one element nil, which is considered as the *witness* of the truth of A, iff A holds. The definition of $\mathbf{x}(A)$ is just the truth value of A in the sense of Fourman 1977.

The type of conjunction $A \equiv A_1 \wedge \cdots \wedge A_n$ represents the Cartesian product or the type of records.

$$\mathbf{x}(A_1) \times \cdots \times \mathbf{x}(A_n) \quad \text{or} \quad \mathbf{prod}(\mathbf{x}(A_1), \ldots, \mathbf{x}(A_n)).$$

But, if A_i is of rank 0 and A_i is false, then the field corresponding to it is omitted. If such a conjunction is used in a scope of an existential quantifier, it turns out a constraint on realizers.

If A, B, C are not of rank 0, then $\mathbf{x}(A \wedge B \wedge C)$ is a three-field record type and, on the other hand, $\mathbf{x}(A \wedge (B \wedge C))$ is a two-field record type. The usual equivalence proof of these two formulas represents the isomorphism between these two records types. As usual $(\wedge I)$ creates a record from fields. But $(\wedge E)$ does more than fetch fields, since it may also introduce conjunctions. This unusual convention is rather useful. For example, we may represent an isomorphism between $\mathbf{x}(A \times B)$ and $\mathbf{x}(B \times A)$ by *just one* application of $(\wedge I)$.

The type of disjunction $A = A_1 \vee \cdots \vee A_n$ is the indexed disjoint sum or the type of alternatives

$$\mathbf{x}(A_1) \amalg \cdots \amalg \mathbf{x}(A_n) \quad \text{or} \quad \mathbf{choose}(\mathbf{x}(A_1), \ldots, \mathbf{x}(A_n)).$$

$(\vee I)$ creates alternatives. $(\vee E)$ is **altcase** construct mentioned in Reynolds 1985. If the disjuncts of the major premise are all of rank 0, then $(\vee E)$ is just the **case** construct.

The type of implication $A \supset B$ is a *subset* of the space $\mathbf{x}(A) \to \mathbf{x}(B)$ of codes of computable functions. It is just $\mathbf{x}(A) \to \mathbf{x}(B)$ as far as $A \supset B$ holds. But it is the empty set, unless $A \supset B$ holds. Hence the constructive axiom of choice (Church's thesis)

$$\forall x.\exists y.A \supset \exists f.\forall x.\nabla y = app{*}(f,x).A$$

cannot be **px**-realized. (Think why this does not conflict with the *rule* of choice.) The inference rules $(\supset E)$ and $(\supset I)$ represent application and abstraction (function closure) as usual. When A is a true rank 0 formula, then $\mathbf{x}(A)$ is a singleton so that

$$\mathbf{x}(A \supset B) \equiv \mathbf{x}(A) \to \mathbf{x}(B) \simeq \mathbf{x}(B).$$

Although we cannot define $\mathbf{x}(A \supset B)$ by $\mathbf{x}(B)$ (see appendix A), it is possible to eliminate the redundant $\mathbf{x}(A)$ in the context of a surrounded universal quantifier. We may refine **px**-realizability of an universal quantifier followed by an implication $A \supset B$ whose assumption part A is of rank 0 so that

$$a\ \mathbf{x}\ \forall : e.A \supset B \quad \text{iff} \quad E(a) \wedge \forall x : e.(A \supset \nabla y = app{*}(a,x).y\ \mathbf{x}\ B).$$

In almost all cases in practice, $\{x : e|A\}$ is a class so that $\forall x : e.A \supset B$ may be replaced by $\forall x : \{x : e|A\}.B$ by which the same refinement is accomplished by the aid of the realizability of bounded universal quantifier. So we do not adopt the above refinement.

The type of universal quantifier is the dependent product of Martin-Löf 1982:

$$\mathbf{x}(\forall x : e.B) \simeq \Pi x : e.\mathbf{x}(B).$$

The inference rules $(\forall E)$ and $(\forall I)$ represent application and abstraction as usual. In the extensional theory of types of Martin-Löf 1982, $\Pi x \in A.B(x)$ is a space of extension of functions, but our space is a space of codes which may not be extensional. Furthermore, in Martin-Löf's theory, A is an arbitrary type and $B(x)$ must be a type that belongs to the same or lower level of types to which A belongs. In our case, A is a "small type" represented by a class and $\mathbf{x}(B(x))$ may be a "large type" represented by a formula. Martin-Löf's type theory has a cumulative hierarchy of types $U_0, U_1, \ldots$ with ω levels; on the other hand, **PX** has only two levels of types, i.e., classes as *small types* and formulas as *large types*. But Martin-Löf's type theory does not have a type expression such as $\mathbf{x}(e_1 : e_2)$ for type expressions e_1, e_2 of the same level, which is our large type expression if e_1 and e_2 are small type expressions. Furthermore, the finite set $\{e_1, \ldots, e_n\}$ is a small type, even if $e_1, \ldots, e_n$ are small types. If the bound variable x is a class variable and e is V, then the type of dependent type is similar to the type $\forall\alpha.\tau$ of the second order λ-calculus. But there is a difference; in second order λ-calculus α ranges over *all* types, but in our case α ranges over only *small* types. Namely, our type quantifier is predicative. So this is more similar to Martin-Löf's type $\Pi x : V_0.A(x)$ of the level V_1, where $A(x)$ is a type of V_1.

Existential quantifier corresponds to the dependent sum. $(\exists I)$ construct a tuple, which is just a list in our language, and $(\exists E)$ is a so-called split, which is the *let* construct in our case. Note that whenever x_1 does not belong to $FV(e_2)$, $\mathbf{x}(\exists x_1 : e_1.\exists x_2 : e_2.A)$ is not only isomorphic but also *identical* to $\mathbf{x}(\exists x_1 : e_1, x_2 : e_2.A)$. On the other hand, $\mathbf{x}(\forall x_1 : e_1, x_2 : e_2.A)$ is not identical, although it is isomorphic modulo *extensional equality* to $\mathbf{x}(\forall x_1 : e_1.\forall x_2 : e_2.A)$. An existential formula with a class variable like $\exists X.A$ is a predicative second order type as similar to the case of the universal quantifier above. Since a realizer of $\exists X.A$ actually carries a witness of X, the dependent sum of the second order existential quantifier is the "strong notion of sum" in the sense of Coquand 1986.

The type of conditional formula $A = e_1 \to A_1; \ldots; e_n \to A_n$ is the type of dependent conditional

$$Cond(e_1, \mathbf{x}(A_1); \ldots; e_n, \mathbf{x}(A_n))$$

defined in 2.4.1. $(\to \vee I)$ and $(\to \wedge E)$ do not express any program constructs. On the other hand, $(\to \vee E)$ and $(\to \wedge I)$ are conditional forms. Note that condition of each clause is an "expression" specified by a user, but the body is a "program" extracted from proofs.

The ∇-quantifier merely specializes the environment of a type expression. But $(\nabla\exists E)$, $(\nabla\forall I)$ represent *let* constructs with arbitrary matching patterns. The other two rules of ∇ do not represent any program constructs.

4.3. CIG recursion

It is well known that the principle of mathematical induction represents primitive recursion. Further correspondence between some structural induction principles and recursions on data structures, such as list induction vs. list recursion, are known. However, the known variety of the recursions represented by induction principles has not been presented in a systematic way, and seems unsuitable to be used as a control structure of an actual programming language. We solve this problem by using the recursion schemata generated by CIG induction. The aim of this section and the following section is to show that such recursion, called CIG recursion, present a wide enough class of recursions to develop recursive programs. The class defined by an instance of CIG inductive definition presents a domain on which the corresponding CIG recursion terminates. This provides a method of separating the termination problem from the partial correctness problem in the framework of "proofs as programs". An implication of this method is that any partial recursive function is programmable by a proof of **PX**, as we will show in 4.4.

Constable and Mendler 1985 have presented similar idea in the context of the type theory of Nuprl. Hagiya and Sakurai 1984 have presented a related idea in the context of verification in logic programming.

4.3.1. Definition of CIG recursion

First, we define recursion schemata called CIG recursions, which are generated from the CIG induction principle. The most general CIG recursions are simultaneous CIG recursions, but we consider only the nonsimultaneous case for simplicity. Let us consider the nonsimultaneous CIG inductive definition

$$\begin{aligned}\textbf{deCIG}\ \vec{x} : C \equiv_D e_1 &\ \rightarrow\ \phi_{1,1}, \ldots, \phi_{1,q_1},\\ &\ \ldots\\ e_n &\ \rightarrow\ \phi_{n,1}, \ldots, \phi_{n,q_n}.\end{aligned}$$

Let $\vec{y}$ be a sequence of individual variables without repetition. For each $i = 1, \ldots, n$ and each $j = 1, \ldots, q_i$, let $\vec{v}_{i,j}$ be a tuple variable whose length equals with the rank of the formula $\phi_{i,j}$. Let $e'_1, \ldots, e'_n$ be arbitrary expressions whose free variables appear among the variables of $\vec{x}$, $\vec{y}$, $FV(\vec{v}_{1,1}), \ldots, FV(\vec{v}_{n,q_n})$. Then the recursion schema of the the following form is called a *CIG recursion* for the above CIG inductive definition:

$$(*)\qquad \begin{aligned} f(\vec{x},\vec{y}) = cond(&e_1, let\ \vec{v}_{1,1} = pred(\phi_{1,1}), \ldots, \vec{v}_{1,q_1} = pred(\phi_{1,q_1})\ in\ e'_1\\ &;\ldots;\\ &e_n, let\ \vec{v}_{n,1} = pred(\phi_{n,1}), \ldots, \vec{v}_{n,q_n} = pred(\phi_{n,q_n})\ in\ e'_n)\end{aligned}$$

An instance of CIG recursion is called a *CIG recursion in the narrow sense* if it is extracted from a proof which ends with CIG induction. Simultaneous CIG recursion is defined in the same way.

The following grammar defines CIG predecessors without the help of CIG templates. So CIG recursion is a logic-free concept.

$$\begin{aligned}\alpha ::= &f(e_1, \ldots, e_n)\\ &|list(\alpha_1, \ldots, \alpha_n)\\ &|cond(e_1, \alpha_1; \ldots; e_n, \alpha_n)\\ &|\Lambda(\lambda(x_1, \ldots, x_n)(\alpha))\\ &|let\ p_1 = e_1, \ldots, p_n = e_n\ in\ \alpha,\end{aligned}$$

where $e_1, \ldots, e_n$ are arbitrary expressions in which f does not appear. For any given recursion of the form $(*)$ whose CIG predecessors are defined by the above grammar, we can give a CIG inductive definition whose CIG recursion is the just the given recursion. One serious restriction of CIG recursion is that $f(\alpha_1, \ldots, \alpha_n)$ is not a CIG template, although $f(e_1, \ldots, e_n)$ is. Consequently CIG recursions tend not to be nested. But by the aid of higher order programming some nested recursions are programmable by means of CIG recursion. We will later show that the Ackermann function is programmable by means of CIG recursion. Another restriction is that the conditions $e_1, \ldots, e_n$ of $(*)$ are expressions in which the function f does not appear. We think this is a *good restriction*, for it makes conditionals simple. Similar restriction is adopted in the regular programming language of Hoare logic.

4.3.2. Examples of CIG recursion

In this subsection, we will present some examples that illustrate how actual recursive Lisp programs are written by means of CIG recursions.

4.3.2.1. Quotient and remainder

The first example is the quotient-remainder algorithm. It is a simple but quite instructive example. The theorem for which we are going to give constructive proofs is the statement

(I) $$\{a : N,\ b : N^+\} \Rightarrow \exists q : N, r : N.(a = b * q + r \wedge r < b).$$

N^+ is the class of positive integers. For readability, we use infix operators such as $*$, $+$, $<$; and some functions, such as $<$, $\geq$, are regarded as both functions and predicates. Assume $e(a, b)$ realizes (I). Then it returns a list (q, r) whose q and r are the quotient and the remainder of the division of a by b insofar as $a : N$ and $b : N^+$. We will give three proofs of this theorem, which represent three different algorithms for computing the quotient and remainder. The fastest one will be presented in 4.6.

In elementary arithmetic, the statement (I) would be proved by mathematical induction on a. We assume that N is defined by CIG as *Nat* of 2.4.1. We assume that the following three sequents are proved:

(A1) $$\{a : N,\ b : N^+, equal(0, a) : T\} \Rightarrow a = b * 0 + 0 \wedge 0 < b,$$

(A2) $$\Gamma \cup \{r + 1 = b\} \Rightarrow a = b * (q + 1) + 0 \wedge 0 < b,$$

(A3) $$\Gamma \cup \{\neg r + 1 = b\} \Rightarrow a = b * q + (r + 1) \wedge r + 1 < b,$$

where

$$\Gamma = \{a : N, b : N^+, equal(0, a) = nil, q : N, r : N, a - 1 = b * q + r \wedge r < b\}.$$

The proof of these sequents do not affect the extracted program, for they are *rank 0 sequents* (sequents with rank 0 conclusions). By the axiom (= 5) we can prove

(A4) $$\{r : N,\ b : N^+\} \Rightarrow r + 1 = b \vee \neg r + 1 = b,$$

and its realizer is

(R1) $$\mathit{cond}(\mathit{equal}(r+1, b), 1; t, 2).$$

Roughly, by the following derivation we can prove (I).

$$\cfrac{\cfrac{(\mathrm{A1})}{S_4}(\exists I) \qquad \cfrac{\{F\} \Rightarrow F \qquad \cfrac{(\mathrm{A4}) \quad \cfrac{(\mathrm{A2})}{S_1}(\exists I) \quad \cfrac{(\mathrm{A3})}{S_2}(\exists I)}{S_3}(\vee E)}{S_5}(\exists E)}{(\mathrm{I})}(CIG\ ind^*)$$

where
$$F \equiv \exists q : N, r : N.(a - 1 = b * q + r \wedge r < b).$$

Its realizer is $f(a, b)$ where f is defined by

$$\begin{aligned}
&f(a,b) = \\
&\quad \mathit{cond}(\mathit{equal}(0,a), \mathit{list}(0,0); \\
&\qquad\quad t, \mathit{let}\ (q\ r) = f(a-1,b)\ \mathit{in} \\
&\qquad\qquad\qquad \mathit{cond}(\mathit{equal}(r+1,b), \mathit{list}(q+1,0); t, \mathit{list}(q, r+1))).
\end{aligned}$$

This program is very slow. When $f(a, b)$ is computed, f is always called a times.

Next we derive a more efficient program by another derivation. Set

$$\textbf{deCIG}\ a : D(b) \equiv_N a < b \rightarrow \top, t \rightarrow a - b : D(b).$$

A natural number a belongs to $D(b)$ iff a eventually becomes smaller than b through successive subtractions by b. Namely, $D(b)$ is the domain on which the Euclidean division algorithm in divisor b terminates. (Note that $D(0)$ is the empty set.) We abbreviate $a = b * q + r \wedge r < b$ by $\phi(a, b, q, r)$. We assume the following two rank 0 sequents have been proved:

(B1) $$\{a : N, b : N, a < b\} \Rightarrow \phi(a, b, 0, a),$$

(B2) $$\{a : N, q : N, r : N, b : N, \phi(a - b, b, q, r), \neg a < b\} \Rightarrow \phi(a, b, q + 1, r).$$

By applying $(\exists I)$ and $(\exists E)$ to $(B2)$, we prove
(B3)

$$\{a : N, b : N, \neg r < b, \exists q : N, r : N.\phi(a - b, b, q, r)\} \Rightarrow \exists q : N, r : N.\phi(a, b, q, r).$$

Let a_1, a_2 be the realizing variables of the formula $\exists q : N, r : N.\phi(a - b, b, q, r)$. Then $list(a_1 + 1, a_2)$ is extracted from $(B3)$ thanks to the optimization done by **subst**. Applying CIG induction on $a : D(b)$ to $\exists q : N, r : N.\phi(a, b, q, r)$ we prove the following sequent from $(B1)$ and $(B3)$:

(B4) $$\{b : N, a : D(b)\} \Rightarrow \exists q : N, r : N.\phi(a, b, q, r).$$

From the proof, $f(a, b)$ is extracted where f is a function defined by

$$f(a, b) = cond(a < b, list(0, a); t, let\ (a_1\ a_2) = f(a - b, b)\ in\ list(a_1 + 1, a_2)).$$

Assuming subtraction and $<$ are primitive functions, this is much more efficient than the previous program. The recursion has the form of iteration in the sense of Backus 1978, and it actually represents the idea of the usual iterative algorithm of division. So far, we have established only the "partial correctness" of f, since $DD = \{(a\ b) | b : N \wedge a : D(b)\}$ is a domain on which f terminates.

We will sometimes use the terminology "partial correctness" in a rather casual sense. Its actual meaning is as follows. In the framework of "proofs as programs" in our setting, a conjecture (or goal formula) gives a specification of a total correctness problem of a program to be extracted from a proof of the conjecture. Let us assume the specification of the function f is that f terminates on each input x from a domain D and x and $f(x)$ satisfies the input-output condition P in the sense of 4.1. Then *the partial correctness problem of the total correctness problem* is: to find a domain E which is a *superset* of D and to prove f terminates for each input x from E and the input and output satisfy P. The problem does *not* include the problem: to *prove* E is a superset of D. Such a problem will be called *the termination problem of the total correctness problem.* If E is just the domain on which f terminates, this problem becomes the partial correctness problem in the usual sense. But in actual proof-program development, this relaxed sense is more useful than the usual sense. Note that we do *not* think of a partial correctness problem alone, but think of a partial correctness part of a given total correctness problem.

So we have finished the partial correctness part of the problem. The next stage is a verification of the termination part. To see that $f(a, b)$ realizes (I), it is sufficient to prove

(B5) $$\{a : N, b : N^+\} \Rightarrow b : N, \quad \{a : N, b : N^+\} \Rightarrow a : D(b).$$

These sequents state that f terminates on the intended domain $\{a, b | a : N \wedge b : N^+\}$. They are easily proved and are rank 0 sequents. By applications of (cut) we finally prove the total correctness statement (I) from the partial correctness statement (B4) and the termination statement (B5). Since the sequents of (B5) are of rank 0, the applications of (cut) do not change the realizer, i.e., $f(a, b)$ is extracted from the proof of (I). In 4.6, we will see how to represent a tail recursive division algorithm by CIG recursion.

4.3.2.2. Maximal element in an integer list

Next we consider the problem to get the maximum element of a nonempty list of integers. The theorem we prove is

(II) $\{a : List_1(N)\} \Rightarrow \exists m : N.(m \in a \wedge \forall x : N.(x \in a \supset x \leq m)).$

The formula $m \in a$ means m is an element of the list a and we suppose it is expressed in a rank 0 formula and $List_1(A)$ is the class of nonempty lists of A defined as in the example (6) of 2.4.1. The most straightforward proof will use list induction. Let $\phi(m, a)$ be the formula

$$m \in a \wedge \forall x : N.(x \in a \supset x \leq m).$$

First we prove the following rank 0 sequents:

(C1) $\Gamma \cup \{fst(a) \geq m'\} \Rightarrow \phi(fst(a), a),$

(C2) $\Gamma \cup \{fst(a) < m'\} \Rightarrow \phi(m', a),$

(C3) $\{a : Dp, snd(a) = nil, fst(a) : N\} \Rightarrow \phi(fst(a), a),$

where Γ is $\{a : Dp, snd(a) : T, a : List_1(N), fst(a) : N, \phi(m', snd(a))\}$. Furthermore we suppose

(C4) $\{fst(a) : N, m' : N\} \Rightarrow fst(a) \geq m' \vee fst(a) < m'$

has been proved, and its realizer is $cond(fst(a) \geq m', 1; t, 2)$. (Such a proof exists.) Apply $(\exists I)$ to (C1) and (C2), then apply $(\vee E)$ to them with (C4) as a major premise. Then a further application of $(\exists E)$ derives
(C5)
$\{a : Dp, snd(a) : T, a : List_1(N), fst(a) : N, \exists m : N.\phi(m, snd(a))\} \Rightarrow \exists m.\phi(m, a).$

Applying $(\exists I)$ to (C3), we prove

(C6) $\{a : Dp, snd(a) = nil, fst(a) : N\} \Rightarrow \exists m.\phi(m, a).$

Then by CIG induction for $List_1$ we can prove (II) from (C5) and (C6). Its realizer is $f(a)$, and f is defined by

$$f(a) = cond(snd(a), let\ b = f(snd(a))\ in\ cond(fst(a) \geq b, fst(a); t, b); t, fst(a)),$$

where b is the realizing variable of $\exists m : N.\phi(m, snd(a))$.

This program is not realistic, for it consumes too much stack space. It is possible to derive a tail-recursive realization of the same sequent. First we define a class

$$\begin{aligned}\mathbf{deCIG}\ [n, a] : M \equiv_{N \times List(N)} & \\ & equal(a, nil) \to \top, \\ & n < fst(a) \to [fst(a), snd(a)] : M, \\ & t \to [n, snd(a)] : M.,\end{aligned}$$

and set

$$\psi(m, n, a) = \Diamond(m = n \vee m \in a) \wedge \forall x : N.((x = n \vee x \in a) \supset x \leq m).$$

Note that $List(N)$ is a list of N including nil. Suppose the following three valid rank 0 sequents:

(D1) $\{equal(a, nil) : T, [n, a] : N \times List(N)\} \Rightarrow \psi(n, n, a),$

(D2) $\Gamma \cup \{\psi(m, fst(a), snd(a)), n < fst(a)\} \Rightarrow \psi(m, n, a),$

(D3) $\Gamma \cup \{\psi(m, n, snd(a)), n \geq fst(a)\} \Rightarrow \psi(m, n, a),$

where Γ is

$$\{equal(a, nil) = nil, \psi(m, fst(a), snd(a)), [n, a] : N \times List(N)\}.$$

Applying $(\exists I)$, $(\exists E)$ and CIG induction to these sequents, we can prove

(D4) $\{[n, a] : M\} \Rightarrow \exists m : N.\phi(m, n, a).$

Its realizer is $f(n, a)$, where f is defined by

$$\begin{aligned}f(n, a) = cond(& equal(a, nil), n; \\ & n < fst(a), f(fst(a), snd(a)); \\ & t, f(n, snd(a))).\end{aligned}$$

This recursive definition is tail recursive and $f(n, a)$ computes the maximum element of the list $pair(n, a)$. By CIG induction for $List_1(N)$, we can prove

(D5) $\{a : List_1(N)\} \Rightarrow [fst(a), snd(a)] : M,$

(D6) $\{a : List_1(N)\} \Rightarrow \forall m : N(\psi(m, fst(a), snd(a)) \supset \phi(m, a)).$

Substitute $\mathit{fst}(a)$ and $\mathit{snd}(a)$ for n and a of (D4). Then the result is

$$\{a : \mathit{List}_1(N)\} \Rightarrow \exists m : N.\psi(m, \mathit{fst}(a), \mathit{snd}(a)).$$

By (*replacement*) with (D6), we can finally prove (II). The realizer extracted from the proof is $f(\mathit{fst}(a), \mathit{snd}(a))$. By the above idea, we will show how to simulates Hoare logic in **PX** in 4.6.

4.3.2.3. The Ackermann function

So far all recursions we derived are not nested. We derive the Ackermann function as an example of a nested CIG recursion through higher order programming. First we define a graph of the Ackermann function.

$$\begin{aligned} &\textbf{deCIG } [x,y,z] : \mathit{Ack} \equiv_{N\times N\times N} \\ &\qquad \mathit{equal}(x,0) \to \mathit{suc}(y) = z, \\ &\qquad \mathit{equal}(y,0) \to [\mathit{prd}(x), 1, z] : \mathit{Ack}, \\ &\qquad t \to \Diamond(\exists z_1 : N.([x, \mathit{prd}(y), z_1] : \mathit{Ack} \land [\mathit{prd}(x), z_1, z] : \mathit{Ack})). \end{aligned}$$

Next we define a class which denotes the recursion used in the Ackermann function.

$$\begin{aligned} \textbf{deCIG } [x,y] : \mathit{Db} \equiv_{N\times N} \ & \mathit{equal}(x,0) \to \top, \\ & \mathit{equal}(y,0) \to [\mathit{prd}(x),1] : \mathit{Db}, \\ & t \to [x, \mathit{prd}(y)] : \mathit{Db}, \forall y : N.[\mathit{prd}(x), y] : \mathit{Db}. \end{aligned}$$

By CIG induction for Db with $(\exists I)$,$(\exists E)$, and $(\forall E)$, we can prove

$$\{[x,y] : \mathit{Db}\} \Rightarrow \exists z : N.[x,y,z] : \mathit{Ack}.$$

Its realizer is $f(x,y)$, where f is defined by

$$\begin{aligned} f(x,y) = \mathit{cond}(&\mathit{equal}(x,0), \mathit{suc}(y); \\ &\mathit{equal}(y,0), f(\mathit{prd}(x),1); \\ &t, f(\mathit{prd}(x), f(x, \mathit{prd}(y)))). \end{aligned}$$

Without the optimization by $\mathit{Optimize}_\beta$, the third clause of the above function definition would be

$$\mathit{app}{*}(\Lambda()(\lambda(y).f(\mathit{prd}(x),y)), f(x,\mathit{prd}(y))).$$

We prove $\{x : N, y : N\} \Rightarrow [x,y] : \mathit{Db}$ by double induction; by (*cut*) we prove

(III) $$\{x : N, y : N\} \Rightarrow \exists z : N.[x,y,z] : \mathit{Ack},$$

and the realizer is the same as the above. The properties of Ack we used in the above derivation, if we replace $[x, y, z] : Ack$ by $Ack(x, y, z)$, are

$$\begin{gathered}
\{equal(x,0)\} \Rightarrow Ack(x,y,suc(y)), \\
\{equal(y,0), x \neq 0, Ack(prd(x),1,z)\} \Rightarrow Ack(x,y,z), \\
\{Ack(x,prd(y),z_1), Ack(prd(x),z_1,z), x \neq 0, y \neq 0\} \Rightarrow Ack(x,y,z),
\end{gathered}$$

where x, y, z, z_1 range over the natural numbers. The above three sequents may be thought as a Prolog program for the the Ackermann function. What we did is to compile it to a deterministic functional program and verify the total correctness of the compiled code.

4.3.2.4. A function whose termination is unknown

The examples so far are functions whose termination are known. In this section we extract a function whose termination is still unknown. The function is

$$\begin{aligned}
\mathbf{def}\ foo(x) = cond(&equal(x,0), 0; \\
&equal(x,1), 1; \\
&even(x), foo(x/2); \\
&t, foo(3*x+1)),
\end{aligned}$$

where $even(x)$ tests if x is an even number. The termination of this function is still unknown, although its termination for many examples has been ascertained by computer experiments. So we do not know how to extract this function with a termination proof, but we *can* extract this function from a proof by the same method used to extract the Ackermann function. Namely we define two CIG classes as follows:

$$\begin{aligned}
\mathbf{deCIG}\ x : D \equiv_N &equal(x,0) \rightarrow \top, \\
&equal(x,1) \rightarrow \top, \\
&even(x) \rightarrow x/2 : D, \\
&t \rightarrow 3*x+1 : D.
\end{aligned}$$

$$\begin{aligned}
\mathbf{deCIG}\ [x,y] : G \equiv_{N\times N} &equal(x,0) \rightarrow [x,0] : G, \\
&equal(x,1) \rightarrow [x,1] : G, \\
&even(x) \rightarrow [x/2,y] : G, \\
&t \rightarrow [3*x+1,y] : G.
\end{aligned}$$

Then it is easy to give a proof of $x : D \Rightarrow \exists y : N.[x,y] : G$ by CIG induction for D. Then from the proof, we can extract the program of foo. What did we prove?

We proved that *foo* terminates on the class D. So the termination problem for *foo* is stated as $\{x : N\} \Rightarrow x : D$.

4.3.2.5. Searching for a prime number

Our technique separating termination and partial correctness provides a way to extract search programs. Finding a prime number which is greater than or equal to a given natural number was a problem for the "proof as program" approach. The problem is formulated as

$$\text{(IV)} \qquad \{n : N\} \Rightarrow \exists p : N.(\mathit{prime}(p) = t \wedge p \geq n),$$

where *prime* is a function which tests if a natural number is a prime. The standard constructive proof of this theorem is as follows.

Assume that there is no prime number in the interval $[n, n! + 1]$. Then every prime which is smaller than or equal to $n! + 1$ is smaller than n. But $n! + 1$ is not divisible by a number m such that $2 \leq m \leq n$. So $n! + 1$ must be a prime number. This is a contradiction. So we see

$$\text{(E1)} \qquad \{n : N\} \Rightarrow \neg\neg\exists m < n! - n + 2.(\mathit{prime}(n + m) = t).$$

On the other hand, since $\mathit{prime}(n + m) = t$ is decidable,

(E2)

$$\{b : N, n : N\} \Rightarrow \exists m < b.(\mathit{prime}(n + m) = t) \vee \neg\exists m < b.(\mathit{prime}(n + m) = t)$$

is constructively provable by mathematical induction on b. Hence we see

$$\{n : N\} \Rightarrow \exists m < n! - n + 2.(\mathit{prime}(n + m) = t),$$

and this implies (IV).

Since mathematical induction is used to prove (E2), the realizer of (E2) uses primitive recursion. The following was extracted by **PX** from a proof of (E2):

```
$f(b, n) =$
    $\mathit{cond}(\mathit{zerop}(b), \mathit{list}(2, \mathit{nil});$
        $t, \mathit{let}\ (a_1\ a_2) = f(b - 1, n)\ \mathit{in}$
            $\mathit{case}(a_1, \mathit{list}(1, a_2), \mathit{cond}(\mathit{prime}(n + b), \mathit{list}(1, n + b); t, \mathit{list}(2, \mathit{nil})))).$
```

The program extracted from the proof of (IV) is

$$\mathit{let}\ (r1\ r2) = f(n! - n + 2, n)\ \mathit{in}\ \mathit{case}(r1, r2, \mathit{quote}(\mathit{dummy})).$$

This calculates $n!$ and calls f recursively $n! - n + 2$ times, even when n is a prime number. So this program is not tractable.

We show how a simple search program is extracted from another proof which uses the proof above as a termination proof.

The program which we expect is a simple search program which looks like

$$\mathit{searchprime}(n) = \mathit{cond}(\mathit{prime}(n), n; t, \mathit{searchprime}(\mathit{suc}(n))).$$

First we define a class which represents this recursion. We call a prime which is greater than or equal to n an upper prime of n. The property "n has an upper prime" can be defined by

$$\begin{aligned}\textbf{deCIG}\ n : \mathit{HasUpperPrime} \equiv_N\ & \mathit{prime}(n) \rightarrow \top, \\ & t \rightarrow \mathit{suc}(n) : \mathit{HasUpperPrime}.\end{aligned}$$

By induction for *HasUpperPrime*, we can prove

$$\text{(E3)} \qquad \{n : \mathit{HasUpperPrime}\} \Rightarrow \exists p : N.(\mathit{prime}(p) = t \wedge p \geq n).$$

Furthermore, we see

$$(*) \qquad \begin{gathered}\{\mathit{prime}(n) = t\} \Rightarrow n : \mathit{HasUpperPrime}, \\ \{m : N, n : N, m \geq n, m : \mathit{HasUpperPrime}\} \Rightarrow n : \mathit{HasUpperPrime}.\end{gathered}$$

So we see,

$$\text{(E4)} \qquad \{n : N, \exists p : N.(\mathit{prime}(p) = t \wedge p \geq n)\} \Rightarrow n : \mathit{HasUpperPrime}.$$

By (E3) and (E4), we see

$$\{n : N, \exists p : N.(\mathit{prime}(p) = t \wedge p \geq n)\} \Rightarrow \exists p : N.(\mathit{prime}(p) = t \wedge p \geq n).$$

Although this sequent looks tautological, the program extracted from the proof above is $\mathit{searchprime}(n)$. Since we have proved (IV), we can prove (IV) from this and the extracted program is again $\mathit{searchprime}(n)$. This proof looks unnatural, since we used a proof of (IV) to prove (IV). To avoid this unnaturalness, we may prove the sequent

$$\text{(E5)} \qquad \{n : N\} \Rightarrow n : \mathit{HasUpperPrime}$$

directly and prove (I) by (E3) and (E5). (E5) is provable by (E1), (E4), and

$$\{\neg\neg n : \mathit{HasUpperPrime}\} \Rightarrow n : \mathit{HasUpperPrime}.$$

(E5) is also provable as (E1) using the two properties $(*)$ of *HasUpperPrime* above.

This method is an application of Markov's principle of 2.3.5. **PX** can prove Markov's principle

$$\diamondsuit \exists x : N.P(x) \supset \exists x : N.P(x)$$

for any provably decidable $P(x)$, and the extracted program is just a search program which tests natural numbers by $P(x)$ successively from zero. So, when one proves existence of x by any method (even classically), one can kill the computational content of the proof by putting $\diamondsuit$ in front and then applying Markov's principle to extract the search program.

4.3.2.6. The Chinese remainder theorem

Many theorems in elementary number theory are constructive and involve various kinds of algorithms. It would be interesting to see how such theorems are formalized by the language of **PX** and such algorithms are represented by CIG recursions. (In this section, we assume a class of integers *Int*, and basic functions and axioms on it. The actual implementation described in chapter 7 has these.)

As a first step in the development of elementary number theory in **PX**, we will do the existence part of the Chinese remainder theorem (see Shockley 1967). The theorem, thousands of years old, is informally stated as follows.

The Chinese remainder theorem. *The system of linear congruences*

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \quad \cdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

has a solution if $m_1, \ldots, m_n$ are pairwise relatively prime.

The first thing to be done is the formalization of the systems of congruences. We formalize a single congruence $a \equiv b \pmod{m}$ by $[a, b] : \mathit{Mod}(m)$, where

$$\mathit{Mod}(m) = \{[a, b] : \mathit{Int} \times \mathit{Int} | m : \mathit{PN} \wedge m : \mathit{Divides}(a - b)\},$$

PN is the class of positive integers and *Divides*(a) is the class of divisors of a defined by

$$\mathit{Divides}(a) = \{a : \mathit{Int} | b : \mathit{Int} \wedge \diamondsuit \exists q : \mathit{Int}.b * q = a\}.$$

A system of congruences $[c, m] : \mathit{SystemOfCong}$ and its solution $x : \mathit{Satisfies}(c, m)$ is formalized as follows:

$$\begin{aligned}\textbf{deCIG}\ [c,m] :& \mathit{SystemOfCong} \equiv \\ & c \to \mathit{fst}(c) : \mathit{Int},\ \mathit{fst}(m) : \mathit{PN},\ [\mathit{snd}(c), \mathit{snd}(m)] : \mathit{SystemOfCong}, \\ & t \to m = \mathit{nil}.\end{aligned}$$

$$\begin{aligned}\textbf{deCIG}\ [c,m] :& \mathit{IsSatisfiedBy}(x) \equiv_{\mathit{SystemOfCong}} \\ & c \to [x, \mathit{fst}(c)] : \mathit{Mod}(\mathit{fst}(m)),\ [\mathit{snd}(c), \mathit{snd}(m)] : \mathit{IsSatisfiedBy}(x), \\ & t \to m = \mathit{nil},\ x : \mathit{Int}.\end{aligned}$$

$$\mathit{Satisfies}(c, m) = \{x | [c, m] : \mathit{IsSatisfiedBy}(x)\}$$

We may formulate the assumption "$m_1, \ldots, m_n$ are pairwise relatively prime" by any rank 0 formula. We will denote it by *PrPrime*(m). Then the Chinese remainder theorem is formulated as

(V) $\{[c, m] : \mathit{SystemOfCong},\ \mathit{PrPrime}(m)\} \Rightarrow \exists x : \mathit{Int}.(x : \mathit{Satisfies}(c, m)).$

Next we prove the following lemma:

Lemma A. *The linear diophantine equation $a * x + b * y = c$ has a solution if the greatest common divisor of a and b divides c.*

The greatest common divisor of a and b will be denoted by $\mathit{gcd}(a, b)$. Our formulation of this lemma is

$$\forall [a, b, c] : \mathit{LemmaAAsp}.\exists x, y : \mathit{Int}.a * x + b * y = c,$$

where *LemmaAAsp* is defined by

$$\mathit{LemmaAAsp} = \{a, b, c | a : \mathit{Int} \wedge b : \mathit{Int} \wedge \mathit{gcd}(a, b) : \mathit{Divides}(c)\}.$$

We prove this lemma by the Euler method (see Shockley 1967) from the following two lemmas:

$$\begin{aligned} \mathit{LemmaB} :&\quad \forall a, b : \mathit{Int}.\exists q : \mathit{Int}, r : N.(a = b * q + r \wedge r : \mathit{AbsLess0}(b)), \\ \mathit{LemmaC} :&\quad \forall b : \mathit{Int}, a : \mathit{Divides}(b).\exists q : \mathit{Int}.a = b * q.\end{aligned}$$

$\mathit{AbsLess0}(b)$ is the class defined by

$$\textbf{deCIG}\ x : \mathit{AbsLess0}(b) \equiv_{\mathit{Int}}\ \mathit{equal}(b, 0) \to x = 0, t \to x < |b|.$$

The former lemma is an extension of the result of 4.3.2.1 and the latter follows from this. We assume these from now on.

We define two auxiliary classes to prove *LemmaA*.

$$\mathit{ImpElm}(a) = \{[a_1, b_1, c_1] : \mathit{LemmaAAsp} | a_1 < a\},$$
$$\textbf{deCIG } [a,b,c] : \mathit{Dom} \equiv_{\mathit{LemmaAAsp}} \forall [a_1, b_1, c_1] : \mathit{ImpElm}(a).[a_1, b_1, c_1] : \mathit{Dom}.$$

Dom denotes the recursion used in the Euler method and *ImpElm* is an example of the technique eliminating an implication mentioned in 4.2.

The partial correctness of the Euler method is stated as

$$\{\mathit{LemmaB}, \mathit{LemmaC}, [a,b,c] : \mathit{Dom}\} \Rightarrow \exists x, y : \mathit{Int}. a * x + b * y = c.$$

This can be proved by induction for *Dom*. In the case $a = 0$, there is y_0 such that $b * y_0 = c$. The solution of the equation is $x = 0$ and $y = y_0$. Let us assume $a \neq 0$. Let q_1, $r1$ and q_2, r_2 be the quotients and remainders when b and c are divided by a. Since r_1 is smaller than a, the equation $r_1 * x + a * y = r_2$ has a solution by the induction hypothesis. Let x_1 and y_1 be a solution. Then $x = q_2 + y_1 - q_1 * x_1$ and $y = x_1$ is a solution of $a * x + b * y = c$.

The termination is stated as

$$\{[a,b,c] : \mathit{LemmaAAsp}\} \Rightarrow [a,b,c] : \mathit{Dom}.$$

The proof is obvious. By these we can prove the lemma by the rule of (*cut*). The program extracted from this proof is $f(a, b, c, \alpha_1, \alpha_2)$, where

```
$f(a, b, c, \alpha_1, \alpha_2) =$
    $cond(equal(a, 0), list(0, app*(app*(\alpha_1, b, c)));$
        $t, let\ (q_2\ r_2) = app*(\alpha_2, c, a)\ in$
           $let\ (q_1\ r_1) = app*(\alpha_2, b, a)\ in$
            $let\ (x_1\ y_1) = app*(\Lambda(\lambda(a_1, b_1, c_1)(f(a_1, b_1, c_1, \alpha_1, \alpha_2))), r_1, a, r_2)\ in$
            $list(q_2 + y_1 - q_1 * x_1, x_1)).$
```

This program has two unnecessary *app∗*'s. This was caused by the limited power of the present version of optimizers. (See appendix C.) The variables α_1 and α_2 are realizing variables of *LemmaC* and *LemmaB*, respectively.

By *LemmaA*, we can prove

$$(*) \quad \begin{aligned} &\forall a, b : \mathit{Int}, m, n : \mathit{PN}.([m, n] : \mathit{RPrime} \\ &\qquad \supset \exists x : \mathit{Int}.([x, a] : \mathit{Mod}(m) \wedge [x, b] : \mathit{Mod}(n))), \end{aligned}$$

by solving the equation $m * x_1 + n * y_1 = a - b$ and taking $n * y_1 + b$ as x. *RPrime* is the class of m and n which are relatively prime.

We prove the Chinese remainder theorem from this fact by the induction for *SystemOfCong*. Assume y_0 is a solution of $[snd(c), snd(m)]$: *SystemOfCong*. Let $\pi(list(a_1, \ldots, a_n))$ be the product $a_1 * \cdots * a_n$. By the assumption of the theorem, $fst(m)$ and $\pi(snd(m))$ are relatively prime. By $(*)$, there is x_0 such that $[x_0, fst(c)] : Mod(fst(m))$ and $[x_0, y_0] : Mod(\pi(snd(m)))$. Obviously $x_0 : Satisfies(c, m)$.

The program extracted from this proof is

```
$g(c, m, \alpha_1, \alpha_2) =$
  $cond(c, app*(\Lambda(m = fst(m),$
                  $n = \pi(snd(m)),$
                  $a = fst(c),$
                  $b = g(snd(c), snd(m), \alpha_1, \alpha_2),$
                  $\alpha_1 = \alpha_1,$
                  $\alpha_2 = \alpha_2)$
             $\lambda()(let\ (x_1\ y_1) = f(m, n, a - b, \alpha_1, \alpha_2)\ in\ n * y_1 + b));$
        $t, 0).$
```

This program computes the product $\pi(snd(m))$ at each time when g is called. To avoid this, we may replace the conclusion of (V) by

$$\exists x, y : Int.(x : Satisfies(c, m) \wedge y = \pi(m)).$$

Then we can use the value of y instead of $\pi(snd(m))$ in the proof of induction step. So π does not appear in the extracted program.

4.4. PX is extensionally complete

We say a class of recursion (recursive definition) is *intensionally complete* when any form of recursive definition belongs to the class. On the other hand, we say a class of recursive functions is *extensionally complete* when any recursive function belongs to the class. (Caution: we use the word "function" in the sense of programming languages. So a recursive function is a partial recursive function in the sense of recursion theory.) Evidently, the class of CIG recursions is not intensionally complete, but it generates an extensionally complete class of functions. Furthermore, any recursive functions can be programmed by CIG recursions in the narrow sense.

We consider recursive functions over natural numbers, i.e., a (partial) function defined by one of the following:

(i) $\phi(x_1,\ldots,x_n)=k \quad (k\in N)$,
(ii) $\phi(x_1,\ldots,x_n)=x_i \quad (i=1,\ldots,n)$,
(iii) $\phi(x)=suc(x)$,
(iv) $\phi(x_1,\ldots,x_n)=\xi(\psi_1(x_1,\ldots,x_n),\ldots,\psi_m(x_1,\ldots,x_n))$,
(v) $\phi(0,x_2,\ldots,x_n)=\delta(x_2,\ldots,x_n)$,
$\phi(x_1,x_2,\ldots,x_n)=\psi(\phi(prd(x_1),x_2,\ldots,x_n),prd(x_1),x_2,\ldots,x_n) \quad \text{if } x_1>0$,
(vi) $\phi(x_1,\ldots,x_n)=\min\limits_{y\in N}(\psi(x_1,\ldots,x_n,y)=0)$.

Let $\phi(x_1,\ldots,x_n)$ be a recursive function. Then we define a class D and a rank 0 formula $G(x_1,...,x_n,y)$ that defines a class. The intended meanings of D and G are the domain and the graph of ϕ. We define G as we defined the graph of a function in 2.4.3. Furthermore we will show

$$\text{(A)} \qquad \forall[x_1,\ldots,x_n]:D.\exists y:N.G(x_1,\ldots,x_n,y)$$

is provable in **PX**. We say that (A) is the specification of ϕ.

The specifications of the functions of (i), (ii), (iii) are

$$\forall[x_1,\ldots,x_n]:N^n.\exists y:N.k=y,$$
$$\forall[x_1,\ldots,x_n]:N^n.\exists y:N.x_i=y,$$
$$\forall x:N.\exists y:N.suc(x)=y.$$

For the composition (iv) we assume $n=1$ for simplicity. Assume that $\forall x_1:D_1.\exists y:N.G_1$ is the specification of ϕ_1 and $\forall x_1:D_2.\exists y:N.G_2$ is the specification of ϕ. Then the specification of $\phi(x_1)$ is

$$\forall x_1:D.\exists y:N.G(x_1,y),$$

where

$$G(x_1,z)=\Diamond y_1:D_1.(G_1(x_1,y_1)\wedge G_2(y_1,z)),$$
$$D=\{x_1:D_1|\exists z:N.G(x_1,z)\}.$$

For (v) we assume $n=2$. Assume that $\forall x_2:D_1.\exists y:N.G_1(x_2,y)$ is the specification of $\delta(x_2)$ and $\forall[y,x_1,x_2]:D_2.\exists z:N.G_2(y,x_1,x_2,z)$ is the specification of

$\psi(y, x_1, x_2)$. Define

$$\begin{aligned}
&\textbf{deCIG } [x_1, x_2, z] : C \equiv \\
&\quad \mathit{equal}(x_1, 0) \to G_1(x_2, z) \wedge x_2 : D_1, \\
&\quad t \to \nabla x_1 = \mathit{prd}(x_1). \\
&\qquad\qquad \Diamond \exists y : N.([y, x_1, x_2] : D_2 \wedge G_2(y, x_1, x_2, z) \wedge [x_1, x_2, y] : C),
\end{aligned}$$

and set

$$G(x_1, x_2, y) = [x_1, x_2, y] : C,$$
$$D = \{x_1 | \Diamond \exists y : N.[x_1, x_2, y] : G\}.$$

Then we can prove (A) for these G and D by mathematical induction on x_1. Note that we cannot prove (A) for these directly from the definition of D, for the following is not constructive:

$$(\Diamond \exists y : N.G) \supset \exists y : N.G.$$

So far only constants, variables, $\mathit{equal}(x, 0)$, and $\mathit{prd}(x)$ are used as expressions. The next step (vi) requires an introduction of a new function using ($\mathit{choice}2$). We assume $n = 1$. Assume $\forall [x, y] : D_1.\exists z : N.G_1(x, y, z)$ is the specification of ψ. Then by the rule of ($\mathit{choice}2$) we may introduce a new function f for which the following is provable:

$$\{[x, y] : D_1\} \Rightarrow \nabla z = f(x, y).G_1(x, y, z).$$

Define

$$\begin{aligned}
\textbf{deCIG } z : D'(x) \equiv_N &\mathit{equal}(f(x, z), 0) \to \top, \\
&t \to \mathit{suc}(z) : D'(x).
\end{aligned}$$

Let $G'(x, y, z)$ be the formula

$$[x, y, z] : N \times N \times N \wedge z \le y \wedge f(x, y) = 0 \wedge \forall u : N.(z \le u < y \supset f(x, u) \ne 0),$$

and set

$$D = \{x : N | 0 : D'(x)\}, \qquad G(x, y) = G'(x, y, 0).$$

Then we can prove the following by induction for $D'(x)$:

$$\{x : N, z : D'(x)\} \Rightarrow \exists y : N.G'(x, y, z).$$

By substituting 0 for z, we can prove (A) for the above G and D, i.e., $\forall x : D.\exists y : N.G(x, y)$ holds.

Hence each partial recursive function $\phi(x_1, \ldots, x_n)$ has a proof P whose conclusion is the form (A) whose D and G satisfy the conditions

$$\begin{gathered}
\mathbf{PX} \vdash \forall [x_1, \ldots, x_n] : D.x_i : N \quad (i = 1, \ldots, n), \\
\mathbf{PX} \vdash \forall [x_1, \ldots, x_n] : D, y : N, y' : N.G(x_1, \ldots, x_n, y) \wedge G(x_1, \ldots, x_n, y') \supset y = y', \\
\mathbf{PX} \vdash \exists y.G(x_1, \ldots, x_n, y) \supset\subset [x_1, \ldots, x_n] : D, \\
G \text{ is the graph of } \phi, \\
D \text{ is the domain of } \phi.
\end{gathered}$$

According to the definition of realizability, insofar as P is a proof of (A) with the above properties, for each $[x_1, \ldots, x_n] : D$,

$$\mathit{extr}(P)(x_1, \ldots, x_n) = \phi(x_1, \ldots, x_n)$$

holds. If we use the recursive definition or a recursion theorem, then the theorem is trivial. To extract a recursive function f, we define f by the aid of recursion, and define D and G as $\{x : N | f(x) : N\}$ and $\{x, y : N \times N | f(x) = y\}$. Then $\forall x : D.\exists y : N.[x, y] : G$ is obvious and f is extracted from the proof. What did we prove then? The point of the theorem is that we used neither recursive definitions nor recursion theorem of **PX** to deduce P for ϕ. We started with basic functions *prd*, *suc*, and *equal*, then we constructed functions by proving theorems in **PX** and extracting functions from such proofs by the rule of (*choice*2). By this accumulation of functions, we could achieve all partial recursive functions. Namely, we programmed recursive functions not by the aid of ordinary programming language but by proofs.

If one wants to verify that a function ϕ which is extracted from a proof P has a property, say

$$(**) \qquad \forall x_1, \ldots, x_n.(\mathit{Input}(x_1, \ldots, x_n) \supset \mathit{Output}(x_1, \ldots, x_n, \phi(x_1, \ldots, x_n))),$$

then it is sufficient to verify the two conditions

$$\begin{gathered}
\mathit{Input}(x_1, \ldots, x_n) \supset [x_1, \ldots, x_n] : D, \\
\mathit{Input}(x_1, \ldots, x_n) \wedge G(x_1, \ldots, x_n, y) \supset \mathit{Output}(x_1, \ldots, x_n, y).
\end{gathered}$$

The first condition says that insofar as the input $x_1, \ldots, x_n$ satisfies the input condition *Input*, ϕ terminates on the input. The second condition says that if ϕ terminates on the input $x_1, \ldots, x_n$ with an output y, then it satisfies the output condition. Hence these are just the conditions of termination and partial correctness of the statement of $(**)$ in the usual sense (not in the casual sense of 4.3). Note that the input and output conditions may be thought to be of rank 0, since they are statements on data, i.e., they do not embody any computational (or constructive) meaning. Hence the above two conditions may be thought to be of rank 0. Thanks to proposition 3 of 2.3.5, one may prove them by virtue of classical logic. Furthermore one may prove them by semantic considerations. This resembles the fact that Cook's relative completeness result on Hoare logic uses formulas that are valid in the intended interpretation of an assertion language as axioms in the consequence rule. Thus we have obtained the following completeness result:

Theorem 1 (extensional completeness of PX). **PX** *can presents any partial recursive function by its proof, and its verification conditions for properties expressed in the language of* **PX** *can also be expressed by the language of* **PX**. *Furthermore, the verification conditions may be proved by any valid reasoning.*

This is a theoretical basis for programming via proofs in **PX**. As noted above, this completeness result strongly resembles the completeness results of Hoare logic. Compare with the above argument with completeness proofs of Apt 1981. We will give a more sophisticated result on relationship between **PX** and Hoare logic in 4.6.

We can program all the partial recursive functions by means of proofs, but it is an open problem whether all *higher order partial recursive* functions can be programmed by proofs. We conjecture that an extension of the theorem to higher order is impossible. But we have not yet been able to show this incompleteness result.

4.5. Transfinite induction

Hayashi 1983 introduced two formulations of the structural induction rule, SIR and SIR_0. These were the main tools for representing recursive programs via proofs in **LM**, an earlier version of **PX**. **PX** does not have those principles as postulates, but CIG induction subsumes them. A difficulty of SIR and SIR_0 was that they were not schemata but rules. This restriction inevitably followed from the fact that our logic does not have variables representing *arbitrary functions*. To state the well-foundness of a binary relation, a variable ranging over arbitrary functions is necessary:

$$\text{(WF1)} \qquad \neg\exists\alpha\forall n : N.R(\alpha(n+1), \alpha(n)).$$

But every function of **LM** and **PX** is a computable function. If α ranges over only computable functions, then the binary relation R is not well-founded even if (WF1) holds. So, to formulate SIR, we had to introduce a new function variable which was intended to range over arbitrary functions and state structural inductions approximately as follows:

$$\text{(SIR)} \qquad \frac{\Gamma \Rightarrow \neg\forall n : N.R(\alpha(n+1), \alpha(n))}{\Gamma \cup \{\forall x((\forall y.R(y,x) \supset \phi(y)) \supset \phi(x))\} \Rightarrow \phi(x)}.$$

The point is that α of the upper sequent is considered to be bound by a universal quantifier, so we could not make it a schema. Although **PX** is a system that does not include the concept of arbitrary functions, by means of the inductive definition via CIG, we can describe the well-foundness of relations by using a single formula without function variables. Let R be a binary relation over a class D, i.e. if $R(x,y)$ holds then $x : D$ and $y : D$. We assume R is a CIG template so that we can define a class Prd by

$$Prd_D(x, R) = \{y : D | R(y, x)\}.$$

Then we define a class Acc of the accessible elements by R as follows:

$$\textbf{deCIG } x : Acc(D, R) \equiv_D \forall y : Prd_D(x, R).y : Acc(D, R).$$

Then R is a well-founded relation on D iff

$$\text{(WF2)} \qquad \forall x : D.x : Acc(D, R).$$

Actually this is the condition of the following rule of induction on R over D:

$$\text{(ID1)} \qquad \frac{\Gamma \cup \{x : D, \forall y : Prd_D(x, R).\phi(y)\} \Rightarrow \phi(x)}{\{x : D\} \cup \Gamma \Rightarrow \phi(x)}.$$

Let us prove this fact. Assume (ID1) holds. Then set

$$\Gamma \equiv \emptyset, \quad \phi(x) \equiv x : Acc(D, R).$$

Then by the definition of Acc the upper sequent of (ID1) holds. So (WF2) holds. On the other hand, by ($CIG\ ind$) for Acc, (ID1) is derived from (WF2) as follows:

$$\frac{\{x : D\} \Rightarrow x : Acc(D, R) \qquad \dfrac{\Gamma \cup \{x : D, \forall y : Prd(x, R).\phi(y)\} \Rightarrow \phi(x)}{\{x : Acc(D, R)\} \cup \Gamma \Rightarrow \phi(x)}}{\{x : D\} \cup \Gamma - \{x : Acc(D, R)\} \Rightarrow \phi(x)}.$$

The expected recursion

$$(*) \qquad f(x, \vec{a}) = g(\Lambda(\lambda(y).f(y, \vec{a})), x, \vec{a}),$$

which was associated with SIR in Hayashi 1983 is actually extracted from this derivation by the extraction algorithm *extr*. Hence SIR of Hayashi 1983 is reformulated as

(SIR$'$) if (WF2) is a theorem in **PX**, then (ID1) is a rule of **PX**.

Note that we may use this principle not only as the rule stated above but also as a schema:

$$\{\forall x : D.x : Acc(D,R)\} \Rightarrow \forall x : D.(\forall y : Prd(x,R).\phi(y) \supset \phi(x)) \supset \forall x : D.\phi(x).$$

Although schematic formulation of SIR is unnecessary for the actual practice of program extraction, its clarity is a theoretical advantage. It is noteworthy that the difference between (WF1) and (WF2) make no difference from a theoretical point of view. By the technique of Hayashi 1982, we can prove that (WF1) is provable in **PX** iff (WF2) is provable in **PX**. Since (WF1) and (WF2) are rank 0 formulas, we may ignore how they are proved. So despite the difference in the formulation of the well-foundness of R, (SIR$'$) may be thought as a reformulation of SIR of Hayashi 1983. Note that the result of Hayashi 1982 establishes only the *provable* equivalence of (WF1) and (WF2), i.e.,

$$\vdash \text{(WF1) iff } \vdash \text{(WF2)}.$$

Since a class defined through a CIG inductive definition represents the minimal fixed point of the inductive definition in the semantics given in chapter 7, (WF2) actually defines the well-foundness of R in our semantics of **PX**. So (WF1) and (WF2) are semantically equivalent, i.e.,

$$\models \text{(WF1)} \supset\subset \text{(WF2)}.$$

We must show how to derive SIR$_0$ and its realizer of Hayashi 1983 from CIG recursion. For simplicity, we do not treat the full formulation, but it is completely straightforward to extend the following arguments to the general case. Assume the following are provable:

$$\begin{gathered}\Gamma \cup \{x : C_1, x : D\} \Rightarrow t_1(x) : D \wedge t_2(x) : D,\\ \Gamma \cup \{x : C_2, x : D\} \Rightarrow t_3(x) : D,\\ \Gamma \cup \{x : D\} \Rightarrow x : C_1 \vee x : C_2,\\ \Gamma \cup \{x : D\} \Rightarrow \neg(x : C_1 \wedge x : C_2),\end{gathered}$$

where Γ is a set of rank 0 formulas. Define $R(y,x)$ by

$$R(y,x) \equiv x : D \wedge \diamondsuit((t_1(x) = y \vee t_2(x) = y) \wedge x : C_1) \vee (t_3(x) = y \vee x : C_2).$$

Further assume that (WF1) or (WF2) holds for these D and R. SIR$_0$ asserts the validity of the following rule under these conditions:

$$\text{(ID2)} \qquad \frac{\begin{array}{c}\Gamma \cup \{x : D, x : C_1, A(t_1(x)), A(t_2(x))\} \Rightarrow A(x) \\ \Gamma \cup \{x : D, x : C_2, A(t_3(x))\} \Rightarrow A(x)\end{array}}{\Gamma \cup \{x : D\} \Rightarrow A(x)}.$$

By the conditions, using ($choice2$), there is an expression e such that

$$\{x : D\} \Rightarrow Case(e, x : C_1, x : C_2)$$

is provable. Define

$$\begin{aligned} deCIG\ x : DD \ \equiv_D\ & equal(e, 1) \to x : C_1, t_1(x) : DD, t_2(x) : DD, \\ & equal(e, 2) \to x : C_2, t_3(x) : DD. \end{aligned}$$

The rule of ($CIG\ ind$) for DD is as follows:

$$\text{(ID3)} \qquad \frac{\Gamma \Rightarrow A \qquad \Pi \Rightarrow A}{\left(\begin{array}{l}\{\ x : DD\ \} \\ \quad \cup\Gamma - \{\ equal(e,1) : T, x : C_1, A(t_1(x)), A(t_2(x))\ \} \\ \qquad \cup\Pi - \left\{\begin{array}{cc} equal(e,1) = nil, & equal(e,2) : T, \\ x : C_2, & A(t_3(x)) \end{array}\right\}\end{array}\right) \Rightarrow A(x)}.$$

By the rule of CIG induction for Acc, we can prove that

$$\{x : Acc(D, R)\} \Rightarrow x : DD.$$

Hence, if WF1 holds for R, then $\forall x : D.x : DD$, so we may replace DD of (ID3) by D. The result is a rule that subsumes (ID2). CIG recursion for (ID3) is

$$f(x, \vec{a}) = case(e, e_0(f(t_1(x, \vec{a}), f(t_2(x, \vec{a}))), \vec{a}), e_1(f(t_3(x), \vec{a}))),$$

which is the realizer associated with SIR$_0$ in Hayashi 1983. In Hayashi 1983 we had to treat SIR and SIR$_0$ as different rules with different realization methods, but now they are special cases of of CIG induction.

The addition of transfinite recursion or transfinite induction for primitive recursive well-orderings on natural numbers are standard proof-theoretic techniques to enlarge proof-theoretic strength and the set of provably recursive functions. The introduction of well-founded relations are also an old technique in program synthesis systems. CIG subsumes these techniques in a natural way, and generalizes them. If we use only transfinite induction for well-founded relations, i.e. SIR$'$

as defined above, the extracted recursion always has the form of $(*)$. But CIG induction can generate a much wider class of recursions.

4.6. Simulating Hoare logic

In this section we present a method by which we can "simulate" Hoare logic in **PX**. The extraction of the maximum element function in 4.3.2.2 turns out to be an example of the technique of this section. We first explain basic motivation for the "simulation".

In light of the technique of developing programs with proofs of correctness of Alagić and Arbib 1978, a derivation in Hoare logic may be thought of not only as a verification of the "written out program" but also as a "trace" or partial "history" of the program's development. In the Hoare logic of a Pascal-like regular programming language, each program construct has exactly one logical inference rule by which the construct is newly introduced. A verified program is determined by the structure of its correctness proof. Hence we can "extract" a program from a derivation of Hoare logic whose programs are hidden. This observation is a key with which to relate the notion of "proofs as programs" and Hoare logic.

Let us illustrate the idea by an example. Consider the following skeleton of a derivation in Hoare logic:

(A)

$$\dfrac{\dfrac{\dfrac{\Pi_1}{\{x \geq y\} \Rightarrow F_1} \quad \dfrac{}{\{F_1\}S_1\{G\}}\,\text{(AA)}}{\{x \geq y\}S_3\{G\}}\,\text{(CR)} \qquad \dfrac{\dfrac{\Pi_2}{\{\neg x \geq y\} \Rightarrow F_2} \quad \dfrac{}{\{F_2\}S_2\{G\}}\,\text{(AA)}}{\{\neg x \geq y\}S_4\{G\}}\,\text{(CR)}}{\{\ \}S_5\{G\}}\,(if\text{R}),$$

where

$$F_1 = x \geq x \wedge x \geq y \wedge x \in \{x, y\},$$
$$F_2 = y \geq x \wedge y \geq y \wedge y \in \{x, y\},$$
$$G = z \geq x \wedge z \geq y \wedge z \in \{x, y\}.$$

and (AA), (CR), (ifR) mean Assignment Axiom, Consequence Rule, and Conditional Rule, respectively. Let us recover the program S_5 from this. The matching of G and F_1 is $z = x$, so, ignoring identical assignment, S_1 must be $z := x$. Similarly we see S_2 is $z := y$. Since consequent rules do not change programs, S_3, S_4 must be S_1, S_2. Similar considerations tell us that S_5 is $if\ x \geq y\ then\ z := x\ else\ z := y$. So the program S_5 is uniquely determined by the structure of logical inferences of the proof skeleton (A).

On the other hand, replacing G by $\exists z.G$, further replacing $\{P\}A\{Q\}$ by $\{P\} \Rightarrow Q$, and adding some trivial sequents, (A) turns into a proof in **PX**:

(B)

$$\dfrac{\Pi_0 \quad \dfrac{\dfrac{\Pi_1}{\{x \geq y\} \Rightarrow F_1} \quad \dfrac{\{F_1\} \Rightarrow F_1}{\{F_1\} \Rightarrow \exists z.G}}{\{x \geq y\} \Rightarrow \exists z.G} \quad \dfrac{\dfrac{\Pi_2}{\{\neg x \geq y\} \Rightarrow F_2} \quad \dfrac{\{F_2\} \Rightarrow F_2}{\{F_2\} \Rightarrow \exists z.G}}{\{\neg x \geq y\} \Rightarrow \exists z.G}}{\{\ \} \Rightarrow \exists z.G},$$

where the last inference is regarded as $(\rightarrow \vee E)$ and Π_0 is a proof of $Sor(x \geq y, \neg x \geq y)$. Furthermore we can reconstruct (A) from (B), and then the realizer of (B) is equivalent to a function that returns the value of the variable z of S_5. Deriving the proof (B) is equivalent to developing the program S_5 with the correctness proof (A). So we say (B) simulates (A). Essentially the same idea appeared in Takasu and Kawabata 1981 and a system based on it has been implemented by Takasu and Nakahara 1983. Their system constructs a Pascal program through an interactive development of a proof of an existence theorem. They used logical inference rules tailored to their specific purpose. In essence, their inference rules are rules of Hoare logic hiding programs. We will show that the inference rules of **PX** can be used for the same purpose, although they were designed to be much more general.

We formulate our simulation technique as the following theorem.

Theorem 1. *Assume the assertion language of Hoare logic is interpretable in* **PX**. *If* $\{Q\}B\{R\}$ *is provable in Hoare logic with all of true statements of the assertion language as axioms, then there is a proof of* **PX** *with true rank 0 sequents as axioms, say* Π, *satisfies the conditions*

(a) *extr*(Π) *is an iterative program, see below, which is equivalent to* B *under* Q.

(b) *The assumption of* Π *is* Q *under the interpretation of* L *in* **PX**.

(c) *The conclusion of* Π *is* $\exists\vec{x}.R$ *under the interpretation of* L *in* **PX**, *where* $\vec{x}$ *is the sequence of variables assigned values by* B.

By the relative completeness of Hoare logic, the provability and validity of $\{Q\}B\{R\}$ are equivalent. So the assumption of this theorem may be "$\{Q\}B\{R\}$ is *valid*" instead of "$\{Q\}B\{R\}$ is *provable*".

PX does not have imperative programming features, so we have to represent them as state transition functions. We say a functional program is an *iterative program* if its recursions involve only tail recursion so that a compiler or an optimizer can *directly* transform them to actual Algol-like imperative programs. The set of iterative programs, say $\mathcal{S}$, is defined by the following. A positive number called a rank is attached to each iterative program (expression).

(1) If $e_1, \ldots, e_n$ are expressions interpreting terms of (the assertion language of) Hoare logic, then $[e_1, \ldots, e_n]$ belongs to $\mathcal{S}$.

(2) Assume e_1 and e_2 are elements of S. Let $\vec{x} = [x_1, \ldots, x_n]$ be a tuple of variables without repetitions such that n is the rank of e_1. Then **subst** $\vec{x} \leftarrow e_1$ **in** e_2 belongs to $\mathcal{S}$ and its rank is the rank of e_2.

(3) If e is an expression interperting a term of the of Hoare logic and $e_1, e_2 \in \mathcal{S}$ with the same rank then $\mathit{cond}(e, e_1; t, e_2)$ is in $\mathcal{S}$ and its rank is the rank of e_1 and e_2.

(4) Assume that e is an expression interpreting a term of Hoare logic and the expressions $e_1, \ldots, e_n$ are elements of $\mathcal{S}$. Let $\vec{x} = x_1, \ldots x_n$ be a sequence of variables without repetition. Then define a function f by the tail recursion

$$f(\vec{x}, \vec{y}) = \mathit{cond}(e, f(e_1, \ldots, e_n, \vec{y}); t, [\vec{x}]).$$

Then $f(\vec{x}, \vec{y})$ is in $\mathcal{S}$ and its rank is n.

These (1)-(4) are counterparts of statements of assignment, composition, conditional, and while, respectively.

A program e of **PX** is said to be equivalent to an imperative program P under a condition Q iff $FV(e)$ is a subset of the set of *all* variables appearing in P, and under the function declaration

$$\textbf{function} \;\; \mathit{foo}(x_1, \ldots, x_n, y_1, \ldots, y_m) \;\textbf{begin}\; P;\; \mathit{foo} := [x_1, \ldots, x_n] \;\textbf{end}$$

the equation

$$e = \mathit{foo}(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

holds under the condition Q, where $x_1, \ldots, x_n$ is the variables to which the program P assigns values and $y_1, \ldots, y_n$ are the rest of the variables appearing in P. We call $x_1, \ldots, x_n$ the *program variables* of P and denote them by $PV(P)$.

This definition of equivalence of the programs guarantees only the extensional equivalence of programs, so the statement of the theorem guarantees only that $\mathit{extr}(\Pi)$ is extensionally equal to B, but the proof of this theorem below constructs a proof Π for which $\mathit{extr}(\Pi)$ is almost intensionally equivalent to B. Namely, if $\mathit{extr}(\Pi)$ is optimized by a clever optimizer that transforms tail recursions to iterations and removes unnecessarily created lists used as tuples, then the optimized $\mathit{extr}(\Pi)$ may be almost literally equivalent to B. Although we do not have a formal description of this fact, the proof below may convince the readers that our claim is reasonable.

We define a Hoare logic of total correctness **HL**, which we are going to simulate by **PX**. Let L be the assertion language of **HL**. We assume there is an interpretation of L in **PX**. The interpretations of expressions of L in **PX** must have values. Especially, true and false must be interpreted as t and nil, respectively. The class of regular programs is defined as usual except that we use simultaneous assignment. We assume that L does not have logical symbols $\exists, \vee$. Since the logic of L is classical, this is not an essential restriction. Furthermore, we assume that $\{\vec{x}|P\}$ is a class in **PX** for any formula P of L. This mild assumption is not necessary but simplifies the proof.

Assignment Axiom :

$$\{P(\vec{t})\}\ \vec{x} := \vec{t}\ \{P(\vec{x})\}.$$

Composition Rule :

$$\frac{\{P\}A\{Q\} \quad \{Q\}B\{R\}}{\{P\}A;B\{R\}}.$$

Conditional Rule :

$$\frac{\{P \wedge e = true\}B_1\{Q\} \quad \{P \wedge e = false\}B_2\{Q\}}{\{P\}\ \mathbf{if}\ e\ \mathbf{then}\ B_1\ \mathbf{else}\ B_2\ \{Q\}}.$$

while Rule :

$$\frac{\{P \wedge e = true \wedge \vec{x} = \overrightarrow{snap}\}B\{P \wedge \overrightarrow{snap} \succ \vec{x}\}}{\{P\}\mathbf{while}\ e\ \mathbf{do}\ B\ \mathbf{od}\{P \wedge e = false\}},$$

where $\overrightarrow{snap}$ is a sequence of "snapshot variables" appearing neither in the program B nor in the formula P and $\vec{x}$ is $PV(B)$, e is a Boolean expression, and the binary relation $\succ$ is well-founded.

Consequence Rule :

$$\frac{P' \supset P \quad \{P\}A\{Q\} \quad Q \supset Q'}{\{P'\}A\{Q'\}}.$$

This system is complete in Cook's sense (see Apt 1981).

Proof of theorem 1. We prove theorem 1 by induction on the structure of proofs of Hoare logic. For simplicity, we assume that the domain of the assertion language is the domain of **PX**.

With the assignment axiom we associate

$$\Pi = \frac{\{P(t_1, \ldots, t_n)\} \Rightarrow P(t_1, \ldots, t_n)}{\{P(t_1, \ldots, t_n)\} \Rightarrow \exists x_1, \ldots, x_n.P(x_1, \ldots, x_n)}(\exists I).$$

Then $extr(\Pi)$ is $[t_1, \ldots, t_n]$, and it can be regarded as an assignment statement of $\mathcal{S}$ satisfying the conditions of the theorem.

Assume that

$$\Pi_1 \vdash \{P\} \Rightarrow \exists \vec{x}'.Q, \quad \Pi_2 \vdash \{Q\} \Rightarrow \exists \vec{x}.R$$

are associated with the premises of the composition rule. Then Π is

$$\frac{\Pi_1 \quad \Pi_2}{\{P\} \Rightarrow \exists \vec{x}.R}{\scriptstyle(\exists E)}$$

and

$$extr(\Pi) = \textbf{subst } \vec{x}' \leftarrow extr(\Pi_1) \textbf{ in } extr(\Pi_2).$$

This belongs to $\mathcal{S}$.

Let Π_1 and Π_2 be associated with the premises of the conditional rule. By the aid of $(\wedge I)$ and (cut) there are Π_1' and Π_2' such that

$$\Pi_1' \vdash \{e : T, P\} \Rightarrow \exists \vec{x}'.Q, \quad extr(\Pi_1') \equiv extr(\Pi_1),$$
$$\Pi_2' \vdash \{e = nil, P\} \Rightarrow \exists \vec{x}''.Q, \quad extr(\Pi_2') \equiv extr(\Pi_2).$$

Then Π is

$$\frac{\begin{array}{c}\Pi_0\\ Sor(e,t)\end{array} \quad \begin{array}{c}\Sigma_1\\ \{P, e : T\} \Rightarrow \exists \vec{x}.Q\end{array} \quad \begin{array}{c}\Sigma_2\\ \{P, e = nil\} \Rightarrow \exists \vec{x}.Q\end{array}}{\{P\} \Rightarrow \exists \vec{x}.Q}{\scriptstyle(\to\vee E)}$$

where $\vec{x}$ is the union of $\vec{x}'$ and $\vec{x}''$, and

$$\Sigma_1 = \frac{\begin{array}{c}\Pi_1'\\ \{e : T, P\} \Rightarrow \exists \vec{x}'.Q\end{array} \quad \dfrac{\{Q\} \Rightarrow Q}{\{Q\} \Rightarrow \exists \vec{x}.Q}}{\{P, e : T\} \Rightarrow \exists \vec{x}.Q}{\scriptstyle(\exists E)},$$

$$\Sigma_2 = \frac{\begin{array}{c}\Pi_2'\\ \{e : T, P\} \Rightarrow \exists \vec{x}''.Q\end{array} \quad \dfrac{\{Q\} \Rightarrow Q}{\{Q\} \Rightarrow \exists \vec{x}.Q}}{\{P, e = nil\} \Rightarrow \exists \vec{x}.Q}{\scriptstyle(\exists E)}.$$

Note that we assume that $Sor(e,t)$ is provable, for e is an expression of **HL**, so it always denotes a value. Then its realizer is

$$cond(e, \textbf{subst } \vec{x}' \leftarrow extr(\Pi_1) \ in \ \vec{x}; t, \textbf{subst } \vec{x}'' \leftarrow extr(\Pi_2) \ in \ \vec{x}).$$

This belongs to $\mathcal{S}$.

Suppose

$$\Sigma \vdash \{P \wedge e = true \wedge \vec{x} = \overrightarrow{snap}\} \Rightarrow \exists \vec{x}.(P \wedge \overrightarrow{snap} \succ \vec{x})$$

is associated with the premise of the **while** rule. Let $\vec{x}$ be $x_1, \ldots, x_n$ and let $extr(\Sigma)$ be d. Set

$$d_i = nth(i, d), \quad \vec{d} = [d_1, \ldots, d_n],$$

where $nth(i, d)$ is the function that fetches the ith element of the list d. By ($choice2$), we may regard $\vec{d}$ as a tuple of expressions defined in **PX**. So we can declare a class $W(\vec{s})$, such that

$$\textbf{deCIG}\ \vec{x} : W(\vec{s}) \equiv e \to \vec{d} : W(\vec{s}), t \to \top.,$$

where $\vec{s} = (FV(\vec{d}) \cup FV(e)) - \{\vec{x}\}$. Set

$$F(\vec{x}) = \exists \vec{x}'.\vec{x} : R(\vec{x}', \vec{s})$$
$$\textbf{deCIG}\ \vec{x} : R(\vec{x}', \vec{s}) \equiv_{\{\vec{x}|P\}} e \to \vec{d} : R(\vec{x}', \vec{s}), t \to \vec{x} = \vec{x}'.$$

Apparently $\{P, e = nil\} \Rightarrow \vec{x} : R(\vec{x}, \vec{s})$ is provable. Let Σ_1' be its proof. Set

(1) $$\Sigma_1 = \cfrac{\begin{array}{c}\Sigma_1' \\ \{P, e = nil\} \Rightarrow \vec{x} : R(\vec{x}, \vec{s})\end{array}}{\{P, e = nil\} \Rightarrow F(\vec{x})}{\scriptstyle (\exists I)}.$$

Then $extr(\Sigma_1)$ is $\vec{x}$. By ($CIG\ def$),

(2) $$\{e : T, P\} \Rightarrow \forall \vec{x}'.(\vec{d} : R(\vec{x}', \vec{s}) \supset \vec{x} : R(\vec{x}', \vec{s}))$$

is provable. Set

$$\Sigma_0 = \cfrac{\{F(\vec{d})\} \Rightarrow F(\vec{d}) \quad (2)}{\{e : T, P, F(\vec{d})\} \Rightarrow F(\vec{x})}{\scriptstyle (replacement)}.$$

Then $extr(\Sigma_0)$ is just the realizing variables of $F(\vec{d})$. By ($CIG\ ind$) we derive a proof Σ such that

$$\Sigma = \cfrac{\begin{array}{c}\Sigma_0 \\ \{e : T, P, F(\vec{d})\} \Rightarrow F(\vec{x})\end{array} \quad \begin{array}{c}\Sigma_1 \\ \{e = nil, P\} \Rightarrow F(\vec{x})\end{array}}{\{\vec{x} : W(\vec{s}), P\} \Rightarrow F(\vec{x})}.$$

Then its realizer is $f(\vec{x}, \vec{s})$ and f is defined by

$$f(\vec{x}, \vec{s}) = cond(e, f(\vec{d}, \vec{s}); t, \vec{x}).$$

This is the **while** statement in the sense of iterative programs $\mathcal{S}$. But this is not the end of the proof. We have to adjust Σ so as to satisfy the conditions (a), (b) of the theorem. By the definition of W, we see from the validity of the premise of the **while** rule that

$$(3) \qquad \{P\} \Rightarrow \vec{x} : W(\vec{s})$$

is a true rank 0 sequent so that we may use it as an axiom. This is the assumption that we may have to assume without a formal proof of **PX**. On the other hand, there is a proof Σ_3 such that

$$\Sigma_3 \vdash \{\ \} \Rightarrow \forall \vec{x}'.(\vec{x} : R(\vec{x}', \vec{s}) \supset (P \wedge e = nil)[\vec{x}'/\vec{x}]).$$

Set

$$\Pi = \cfrac{\cfrac{(3) \quad \Sigma}{\{P\} \Rightarrow F(\vec{x})}{\scriptstyle (cut)} \qquad \Sigma_3}{\{P\} \Rightarrow \exists \vec{x}.(P \wedge e = nil)}{\scriptstyle (replacement)\&(alpha)}.$$

Then $extr(\Pi) = extr(\Sigma)$ and Π satisfies the conditions of the theorem.

The counterpart of the consequence rule is just (*replacement*). It does not change the program. This ends the proof of theorem 1. □

What we have proved can be summarized by the diagram

$$\begin{tikzcd}
\Sigma \vdash_{\mathbf{HL}} \{P\}A\{Q\} \arrow[r] \arrow[d] & \Pi \vdash_{\mathbf{PX}} \{P\} \Rightarrow \exists \vec{x}.Q \arrow[d] \\
A \arrow[r, phantom, "\cong"] & extr(\Pi)
\end{tikzcd}$$

The correspondence between **HL** and **PX** used in the above proof is summarized by the table

HL	**PX**
Assertion formula	Rank 0 formula
$\Sigma \vdash \{P\}A\{Q\}$	$\Pi \vdash \{P\} \Rightarrow \exists \vec{x}.Q$
Assignment	$(\exists I)$
Composition	$(\exists E)$
Conditional	$(\rightarrow \vee E)$
While	$(CIG\ ind)$
Consequence	(*replacement*)

As an example of the above method, we derive another program of the quotient-remainder theorem (I) of 4.3.2.1. Set

$$D(a,b) = \{q,r|q : N \wedge r : N \wedge a = b * q + r\}$$

and

$$\textbf{deCIG } [x,y] : W(a,b) \equiv_{D(a,b)} y \geq b \rightarrow [x+1, y-b] : W(a,b), t \rightarrow \top.$$
$$\textbf{deCIG } [x,y] : R(q,r,a,b) \equiv_{D(a,b)} y \geq b \rightarrow [x+1, y-b] : R(q,r,a,b),$$
$$t \rightarrow x = q, y = r.$$

As in the proof of the theorem, we can derive a program Σ such that

(4) $\quad \Sigma \vdash \{a : N, b : N^+, [x,y] : W(a,b)\} \Rightarrow \exists q : N, r : N.[x,y] : R(q,r,a,b)$

and $extr(\Sigma)$ is $f(x,y,a,b)$ whose f is defined by

$$f(x,y,a,b) = cond(y \geq b, f(x+1, y-b, a, b); t, list(x,y)).$$

Substitute $0, a$ for x, y in (4). Since $[0,a] : W(a,b)$ holds provided $a : N, b : N^+$, we can eliminate this from the assumption of (4). Since R is inductively defined, if $R(q,r,a,b)$ is nonempty, then there is a value $[x_0, y_0]$ of $[x,y]$ satisfying the conditions of the base case:

$$[x_0, y_0] : D(a,b),\ y_0 < b, x_0 = q,\ y_0 = r.$$

So $a = b * q + r \wedge r < b$ holds. Formalizing this argument, we can derive

(5) $\quad \forall q : N, r : N.([0,a] : R(q,r,a,b) \supset a = b * q + r \wedge r < b).$

Hence the sequent (I) of 4.3.2.1 is derivable by (*replacement*) and its realizer is the expression $f(0,a,a,b)$. This is a tail-recursive functional program for the Euclidean division algorithm.

5 PX as a foundation of type theories

There are quite a few type theories of programming languages (see Cardelli and Wegner 1986). In this chapter, we will show how some type theories are interpreted in **PX**. The subjects we consider here are (i) dependent sums as types of modules and (ii) polymorphic types in the sense of second order λ-calculus. The interpretation of subtypes as subsets is straightforward, since **PX** is a type free system; and interpretation of the types in the sense of Martin-Löf 1982 is possible as was showed in 2.4. Note that these type disciplines are interpreted in **PX**, so it is child's play to mix them in the single type free framework of **PX**.

5.1. A view of "dependent sums as types of module implementations"

The dependent sum was used as the type of the module implementations in Burstall and Lampson 1984. To form a type of the implementations of a module, they used the type of all types used in implementation. (The type of all types is an element of itself.) If such a type were a class, say $CLASS$, i.e., the class of all classes, then it would be possible to define a type of implementations of stack module function (stack module with a parameter x) such as

$$\Pi(CLASS, \lambda(x).\Sigma(CLASS, \lambda(y).(x \times y \rightarrow y) \times (y \rightarrow x \times y) \times y).$$

Unfortunately, the existence of $CLASS$ contradicts the axiom system of **PX** (see 6.3.1), although it is consistent with $\mathbf{PX} - \{(\mathit{Join}), (\mathit{Product})\}$ as shown in 6.3. But CIG allows us to define the class of classes generated by given finite numbers of classes and class formation operators as in (12) of 2.4.1. For example, the class of implementations of a module implemented in a class of finitely generated CCC is obtained by replacing $CLASS$ by $CCC(X_1, \ldots, X_n)$ and $a \rightarrow b$ by $\mathit{hom}(a, b)$. If we add $\mathit{List}(X)$ and $X + Y$ in the possible class formation operators of $CCC(X_1, \ldots, X_n)$, then the class of finitely generated classes, say $\mathit{Types}_0(X_1, \ldots, X_n)$, would be enough to implement conventional modules such as stack, queue, records, etc., insofar as $X_1, \ldots, X_n$ includes enough basic data types. Since $\mathit{Types}_0(X_1, \ldots, X_n)$ is again a class, a higher type such as the type of module functions, i.e., functions mapping implementations of a module to another implementation of module, is again a class. The class $\mathit{Types}_0(X_1, \ldots, X_n)$ is not an element of itself, so we have a hierarchy of modules as the module system of MacQueen 1986. The essential difference of our module hierarchy to MacQueen's is that each stage of our hierarchy is not closed under dependent type formations. Since dependent types are used as types of modules or module functions, this does not seem to be a serious restriction.

One can include various "logical" conditions on the type of the implementations of a module. For example, a description of the class of implementations of stack module functions would be

$$\begin{aligned}
&\mathit{Stck}(x,\vec{X}) = \Sigma(\mathit{Types}_0(\vec{X}), \lambda(y).(\mathit{hom}(x \times y, x) \times \mathit{hom}(y, x \times y) \times y)),\\
&f = \lambda(x).\{(y\ (\mathit{pu}\ \mathit{po}\ \mathit{empty})) : \mathit{Stck}(x,\vec{X})|\\
&\qquad \forall b : x, s : y.(\mathit{pop}(\mathit{push}(b,s)) = \mathit{list}(b,s)\\
&\qquad\qquad \wedge\, \mathit{pop}(\mathit{empty}) = \mathit{list}(\mathit{nil}, \mathit{empty}))\}\\
&\mathit{Stack}(\vec{X}) = \Pi(\mathit{Types}_0(\vec{X}), f),
\end{aligned}$$

where $\vec{X}$ is $X_1, \ldots, X_n$ and

$$\mathit{push} = \lambda(b,s).\mathit{app}{*}(\mathit{pu}, b, s), \quad \mathit{pop} = \lambda(s).\mathit{app}{*}(\mathit{po}, s).$$

The body of the λ-function of f is not a legal class expression, for y appears as $s : y$. Let us explain how to validate such a class expression. Let p be a pattern and $y_1, \ldots, y_n$ be some free variables of p such that

$$\forall x : A.\nabla p = x.(\mathit{Class}(y_1) \wedge \ldots \wedge \mathit{Class}(y_n)).$$

Let ϕ be a CIG template with respect to $X_0, X_1, \ldots, X_{m+n}$. We assume that X_0 does not appear in ϕ. This assumption is made for the sake of simplicity. Then we define an extended class expression as follows:

$$\begin{aligned}
&\{p : A|\phi[y_1/X_1, \ldots, y_n/X_n]\} = \{x : A|\mathit{pair}(x, \mathit{nil}) : B\},\\
&B = \Sigma(A, \lambda(x).\mathit{let}\ p = x\ \mathit{in}(\{r|r = \mathit{nil} \wedge \phi\}[y_1/X_1, \ldots, y_n/X_n])).
\end{aligned}$$

Then it is easy to see that $x : \{p : A|\phi[y_1/X_1, \ldots, y_n/X_n]\}$ holds iff $x : A \wedge \nabla p = x.\phi$.

If we use the second order extension of **PX** in 5.2, we can define a class of stack functions satisfying an initial algebra condition by replacing *Stck* of the definition of *Stack* by

$$\begin{aligned}
&\mathit{InitStck}(x,\vec{X}) =\\
&\quad \{(y\ (\mathit{pu}\ \mathit{po}\ \mathit{empty})) : \mathit{Stck}(x,\vec{X})|\\
&\qquad \forall X.((\mathit{empty} : X \wedge \forall b : y, s : X.\mathit{push}(b,s) : X) \supset y \subseteq X)\}.
\end{aligned}$$

Furthermore, by means of logical inferences of **PX**, we can derive various kinds of properties of modules.

Pebble in Burstall and Lampson 1984 has values called bindings and their types. A binding is a tuple of tagged value or, more precisely, an environment as a value. If x is an identifier and v is a value, then $x{\sim}v$ is a binding and it expresses the environment in which v is bound to x. Multiple binding is available, and is written as $[x_1{\sim}v_1, \ldots, x_n{\sim}v_n]$. The type of this binding is written as $x_1 : A_1 \times \ldots \times x_n : A_n$, where $A_1, \ldots, A_n$ are types of $v_1, \ldots, v_n$. Since **PX** cannot handle environments as values, it does not have bindings in the sense of Burstall and Lampson 1984. But the main usage of binding is as an implementation of modules. This is interpretable in **PX**. A module of $Stck(x)$ is just a list with four elements, say $(x_1\ x_2\ x_3\ x_4)$. When we use bingings, we may write $[stack{\sim}x_1, push{\sim}x_2, pop{\sim}x_3, empty{\sim}x_4]$. This makes the meaning of elements clear. More, it allows the mechanism of inheritance. Namely, even if the type of input of a module is a type of bindings of the form $[x{\sim}value]$, it can be applied to a binding which extends this form, e.g., $[x{\sim}value, y{\sim}value]$. The inheritance of Pebble is accomplished by a coercion. When a module function is applied to a multiple binding that has excess bindings, it is shrunk down to the proper shape. Namely, when a function f from $x : A$ is applied to an element of $x : A \times y : B$, say b, $f(b)$ is automatically interpreted as $f(proj_1(b))$. On the other hand, **PX** may support inclusion polymorphism. Namely, we may interpret the class of bindings so that $x : A \times y : B$ is a *subtype* of the type of $x : A$. This can be achieved by interpreting the type of bindings as follows. Let $Bndg$ be the class

$$\{b : List(Atm \times V) | \forall m : N, n : N.m \neq n \supset fst(nth(m, b)) \neq fst(nth(n, b))\},$$

where $nth(n, a)$ is n th element of a list a. This is a class of all bindings. A class that represents the type $x_1 : A_1 \times \ldots \times x_n : A_n$ is defined by

$$\{b : Bndg | fetch(x_1, b) : A_1 \wedge \ldots \wedge fetch(x_n, b) : A_n\},$$

where $fetch(x, b)$ returns the element of b whose fst part is x, if it exists.

The module system IOTA of Nakajima and Yuasa 1983 does not have the type of all types, but it has a verification system by which users can verify that an implementation meets its specification. IOTA allows arbitrary first order formulas in specifications, and initial algebra conditions may be put in the specifications. Furthermore, any module is built up from fixed (so finitely many) built-in basic parameterized or nonparameterized modules. It seems that modules of IOTA are enough for conventional programming. (See the examples given in Nakajima and Yuasa 1983.) It is likely that any practical module system can be built on finite many basic modules as in IOTA. The technique described above may serve as a logical foundation of IOTA-like finitely generated module systems and may be enough to cover a large portion of module building activity in conventional programming.

5.2. Type polymorphism and an impredicative extension of PX

The type polymorphism is a useful discipline in typed languages. We will show how polymorphic types are interpreted as classes. To this end we extend the CIG inductive definition so that *impredicative* definition of classes is possible. The CIG inductive definition is *predicative* definition of classes, for bound class variables are inhibited in CIG templates. This means all classes are gradually constructed from basic classes by applications of explicit class formation operators. Actually such processes will be observed in the model constructions in 6.3. On the other hand, impredicative classes must be given at once as the power sets of infinite sets. Templates for impredicative CIG inductive definition, called stratified templates, are defined as follows:

Definition 1 (stratified template). A formula H is a stratified template of $Str_{n_0}(\vec{X})$ iff it satisfies the following conditions:

(1) If a subformula of H has the form $[e_1, \ldots, e_n] : e_{n+1}$, then e_{n+1} must be a class constant or a bound class variable or belong to $\vec{X}$. Furthermore, if e_{n+1} is X_0, then the entire subformula must occur in a positive part of H and n is n_0.If a subformula of H has the form $E(e)$, then e has no occurrence of X_0. If a subformula of H has the form $e_1 = e_2$, then e_1 and e_2 have no occurrence of X_0.

(2) For any subformula of H of the form $\forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.F$ or $\exists \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.F$, the following holds: When there is a class variable in the variables of $\vec{x}_i$, e_i is the class constant V. Otherwise, e_i is an element of $\vec{X}$ or a class constant or a bound class variable. Similarly for the existential quantifier.

(3) H has no occurrence of the predicate symbol $Class$.

(4) In any subformula of the form $E(e)$, $e_1 = e_2$, there are no bound class variables. There are no bound class variables in $[e_1, \ldots, e_n]$ of any subformula of the form of $[e_1, \ldots, e_n] : e$.

Any subformula of a stratified template is called a *stratified formula*. We extend CIG allowing the stratified templates as CIG templates. Then the extended CIG inductive definition is called *CIG2 inductive definition*. Note that CIG2 is an extension of CIG in two ways. One thing we do not assume is that a CIG2 template is a rank 0 formula. The other is that a CIG2 template may contain bound class variables. The second assumption is an essential extension, but the first one is not, for we do not consider realizability of CIG2. (Feferman 1979 described a realizability for an impredicative extension of T_0, but it turned out that it does not work properly, for the realizability of a stratified formula may not be stratified. So the existence property and consistency with Church's thesis of the system are still unproved.)

A polymorphic function is a typed function that may apply to different types uniformly. For example, *append* is applicable to any type of the form $List(X) \times List(X)$. We will give so-called "forgetful semantics" of an extensional theory of second order type polymorphism in **PX** + CIG2. Such a semantics of type polymorphism was first given by Girard 1972 and Troelstra 1973 in the framework of ordinary recursion theory, and later MacQueen and Sethi 1982 gave essentially the same semantics in the framework of Scott semantics. Feferman has given essentially the same semantics to ours in an impredicatively extended T_0.

First we define a typing system PT (polymorphic typing). PT has three syntax categories called types, terms, and judgment. A *type*, or *type expression*, is an expression that is inductively defined as follows:

1. A type variable is a type. We denote type variables by α, β, etc. On the other hand, types will be denoted by τ, σ, etc.
2. If τ and σ are types, then $\tau \to \sigma$ is a type.
3. If τ is a type and α is a type variable, then $\forall\alpha.\tau$ is a type.

We call $\forall\alpha.\tau$ a polymorphic type and free type variables of types, etc. are defined as usual.

A *term* is simply a λ-term in the sense of λ-calculus, i.e.,

1. Variables are terms. Variables will be denoted by x, y, etc., and terms will be denoted by t, s, etc. (We assume that such variables are disjoint to the type variables.)
2. If t and s are terms, then its application ts is a term.
3. If t is a term and x is a variable, then its abstraction $\lambda x.t$ is a term.

As opposed to the usual typed λ-calculus, we do *not* identify two α-convertible terms or types implicitly. α-convertibility of types is formulated as inference rules. On the other hand, α-convertibility of terms is derivable from the others. A *judgment* is an expression of the form $t : \tau$ or $t_1 =_\tau t_2$. Note that any syntactic expression of this form is called judgment, even if it is not provable.

An *environment* is a partial function from variables to types with finite domain. It is denoted by Γ, Δ, etc.. An environment Γ with a domain $\{a_1, \ldots, a_n\}$ is also denoted by a set $\{a_1 : \tau_1, \ldots, a_n : \tau_n\}$, where $\tau_i = \Gamma(a_i)$. $\Gamma, x : \tau$ means the extension of the environment Γ that maps x to τ. So we assume x does not belong to the domain of Γ. A *hypothetical judgment* is a form $\Gamma \vdash A$, where Γ is an environment and A is an judgment.

Next we define *typing*, i.e., provable hypothetical judgments or theorems. Typings are generated by the axiom

$$\text{(A1)} \qquad \{a_1 : \tau_1, \ldots, a_n : \tau_n\} \vdash a_i : \tau_i$$

and the following rules:

(R1) $\dfrac{\Gamma \vdash t : \tau}{\Gamma \vdash t =_\tau t}$ (R2) $\dfrac{\Gamma \vdash t =_\tau t}{\Gamma \vdash t : \tau}$

(R3) $\dfrac{\Gamma \vdash t_1 =_\tau t_2}{\Gamma \vdash t_2 =_\tau t_1}$ (R4) $\dfrac{\Gamma \vdash t_1 =_\tau t_2 \quad \Gamma \vdash t_2 =_\tau t_3}{\Gamma \vdash t_1 =_\tau t_3}$

(R5) $\dfrac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$ (R6) $\dfrac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau}$

(R7) $\dfrac{\Gamma \vdash t : \tau}{\Gamma \vdash t : \forall \alpha.\tau}$ (R8) $\dfrac{\Gamma \vdash t : \forall \alpha.\tau}{\Gamma \vdash t : \tau[\sigma/\alpha]}$

(R9) $\dfrac{\Gamma, x : \sigma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash (\lambda x.t_1)(t_2) =_\tau t_1[t_2/x]}$ (R10) $\dfrac{\Gamma \vdash t : \sigma \to \tau}{\Gamma \vdash \lambda x.(tx) =_{\sigma \to \tau} t}$

(R11) $\dfrac{\Gamma, x : \sigma \vdash t_1 =_\tau t_2}{\Gamma \vdash (\lambda x.t_1) =_{\sigma \to \tau} (\lambda x.t_2)}$ (R12) $\dfrac{\Gamma \vdash t_1 =_{\sigma \to \tau} t_2 \quad \Gamma \vdash t_3 =_\sigma t_4}{\Gamma \vdash t_1 t_3 =_\tau t_2 t_4}$

(R13) $\dfrac{\Gamma \vdash t_1 =_\tau t_2}{\Gamma \vdash t_1 =_{\forall \alpha.\tau} t_2}$ (R14) $\dfrac{\Gamma \vdash t_1 =_{\forall \alpha.\tau} t_2}{\Gamma \vdash t_1 =_{\tau[\sigma/\alpha]} t_2}$

(R15) $\dfrac{\{a_1 : \tau_1, \ldots, a_n : \tau_n\} \vdash t : \tau_0}{\{a_1 : \tau_1', \ldots, a_n : \tau_n'\} \vdash t : \tau_0'}$ (R16) $\dfrac{\{a_1 : \tau_1, \ldots, a_n : \tau_n\} \vdash t_1 =_{\tau_0} t_2}{\{a_1 : \tau_1', \ldots, a_n : \tau_n'\} \vdash t_1 =_{\tau_0'} t_2}$

These rules are legal under the following conditions: in (R5), (R9), and (R11) the variable x does not appear as a free variable in Γ, in (R7) and (R13) the variable α does not appear as a free type variable in Γ, in (R8), (R9), and (R14) $\tau[\sigma/\alpha]$ and $t_1[t_2/x]$ are legal substitutions, e.g., t_2 must be substitutable to x without any change of bound variables of t_1, in (R10) x is not a free variable of t, and in (R15) and (R16) τ_i is α-convertible to τ_i'.

Note that if $\Gamma \vdash t_1 =_\tau t_2$ is provable, then $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$ are provable by (R2), (R3), and (R4). If $\Gamma \vdash t : \tau$ is provable and t' is α-convertible to t', then $\Gamma \vdash t' : \tau$ is also provable. This can be proved by induction on the length of the derivation of typing. It is not quite trivial, but is left for the reader.

The semantics of PT that we introduce below is known as HEO$_2$ (Troelstra 1973) or PER model (Mitchell 1986). Our semantics is a "formalized semantics" in the sense that it is a syntactic translation from PT to **PX**+CIG2. We translate each typing to a provable sequent of **PX**+CIG2, by translating terms to expressions, types to classes, and judgments to formulas. We assume there are one to one mappings i and j from the variables of PT to (total) individual variables and type variables to (total) class variables. We assume they coincide on the type variables, but differ on the variables. We denote $i(a)$ by $\bar{a}$ and $j(a)$ by $\tilde{a}$. So $\bar{\alpha} = \tilde{\alpha}$ for any type variable α, but $\bar{x} \neq \tilde{x}$ for any variable x. For each type τ, we define a class expression $\bar{\tau}$ and Eq_τ, which are interpretated to be the type τ and the equality of the type. Since Eq_τ is an equality defined over $\bar{\tau}$, it is a *partial equivalence relation*, i.e., $\forall [a,b] : Eq_\tau . [b,a] : Eq_\tau$ and $\forall [a,b] : Eq_\tau, [b,c] : Eq_\tau . [a,c] : Eq_\tau$ hold, but $\forall a . [a,a] : Eq_\tau$ need not hold. The domain of any partial equivalence relation $R(x,y)$ is defined by $\{x | R(x,x)\}$, i.e., the largest set on which R is an equivalence relation. Hence a type equipped with an equivalence relation on it is realized as a partial equivalence relation. So the association of the interpretation of types is done as follows:

0. For each type τ, $\bar{\tau}$ is $\{x | [x,x] : Eq_\tau\}$.
1. For a type variable α, Eq_α is $\bar{\alpha}$.
2. $Eq_{\sigma \to \tau}$ is given by

$$[x,y] : Eq_{\sigma\to\tau} = \{x,y | \forall [a,b] : Eq_\sigma . [app{*}(x,a), app{*}(y,b)] : Eq_\tau\}.$$

3. $Eq_{\forall\alpha.\tau}$ is given by

$$[x,y] : Eq_{\forall\alpha.\tau} = \{x,y | \forall \bar{\alpha} . PER(\bar{\alpha}) \supset [x,y] : Eq_\tau\},$$

where $PER(\bar{\alpha})$ is a formula which stands for $\bar{\alpha}$ is a partial equivalence relation (see above).

Next, we define expressions $\bar{t}$ and $\tilde{t}$ for each term t.

1. For a variable x, $\bar{x}$ is $\bar{x}$.
2. The interpretation of abstraction $\overline{\lambda x.t}$ is $\Lambda(\lambda(\bar{x})(\bar{t}))$.
3. The interpretation of application $\overline{t_1 t_2}$ is $app{*}(\bar{t}_1, \bar{t}_2)$.
4. The interpretation $\tilde{t}$ is defined as above, except a variable x is translated to $\tilde{x}$ instead of $\bar{x}$.

The interpretations of $t : \tau$ and $t_1 =_\tau t_2$ are

$$[\bar{t}, \tilde{t}] : Eq_\tau$$

and

$$[\bar{t}_1, \tilde{t}_1] : Eq_\tau \wedge [\bar{t}_2, \tilde{t}_2] : Eq_\tau \wedge [\bar{t}_1, \tilde{t}_2] : Eq_\tau,$$

respectively. We denote the interpretation of a judgment A by $\bar{A}$. The interpretation of an environment $\Gamma = \{a_1 : \tau_1, \ldots, a_n : \tau_n\}$, $\bar{\Gamma}$ in notation, is defined as follows:

$$\bar{\Gamma} = PER_\Gamma \cup \{[\bar{a}_1, \tilde{a}_1] : Eq_{\tau_1}, \ldots, [\bar{a}_n, \tilde{a}_n] : Eq_{\tau_n}\},$$
$$PER_\Gamma = \{PER(\bar{\alpha}) | \alpha \text{ is a free type variable in } \Gamma\}.$$

Furthermore, we define

$$\bar{\Gamma}_A = \bar{\Gamma} \cup PER_A,$$
$$PER_A = \{PER(\bar{\alpha}) | \alpha \text{ is a free type variable of } A\}.$$

The interpretation of a hypothetical judgment $\Gamma \vdash A$ is given by $\bar{\Gamma}_A \vdash \bar{A}$.

Then we can prove the following theorem by the induction on the length of derivations of typings.

Theorem. *If $\Gamma \vdash A$ is a typing, then $\bar{\Gamma}_A \vdash \bar{A}$ is provable in* **PX**+*CIG2.*

As a corollary we have the following:

Corollary. *Set*

$$\hat{\Gamma}_A = \{\bar{a} : \bar{\tau} | a : \tau \in \Gamma\} \cup PER_\Gamma \cup PER_A,$$
$$\widehat{t : \tau} = \bar{t} : \bar{\tau}, \qquad \widehat{t_1 =_\tau t_2} = [\bar{t}_1, \bar{t}_2] : Eq_\tau$$

If $\Gamma \vdash A$ is a typing, then $\hat{\Gamma} \vdash \hat{A}$ is provable in **PX**+*CIG2.*

Hence there is no term t such that $\vdash t : \forall\alpha.\alpha$ is a typing. Namely, PT is consistent in the sense of traditional logic. (Even if such a term exists, a system may be consistent from the view point of computation. Since we may think the value of such a term is an "undefined value".)

Proof of theorem. The theorem is proved by the induction of the complexity of the derivation of typings. Except for the verification of (R9), nothing is difficult. Let us show how to verify (R9). The induction hypotheses are

$$\bar{\Gamma} \cup \{[\bar{x}, \tilde{x}] : Eq_\sigma\} \cup PER_\sigma \cup PER_\tau \Rightarrow [\bar{t}_1, \tilde{t}_1] : Eq_\tau,$$

$$\bar{\Gamma} \cup PER_\sigma \Rightarrow [\bar{t}_2, \tilde{t}_2] : Eq_\sigma,$$

where PER_τ is the set $\{PER(\bar{\alpha}) | \alpha \in FV(\tau)\}$. We have to show the following three are provable under the assumption $\bar{\Gamma} \cup PER_\tau$:

$$(1)\ [\overline{(\lambda x.t_1)t_2}, \widetilde{(\lambda x.t_1)t_2}] : Eq_\tau, \quad (2)\ [\overline{t_1[t_2/x]}, \widetilde{t_1[t_2/x]}] : Eq_\tau,$$
$$(3)\ [\overline{(\lambda x.t_1)t_2}, \widetilde{t_1[t_2/x]}] : Eq_\tau.$$

Let us verify (3). $\overline{(\lambda x.t_1)t_2}$ is $app{*}(\Lambda(\lambda(\bar{x})(\bar{t}_1)), \bar{t}_2)$. We may assume $E(\bar{t}_2)$ by the induction hypothesis. So this is equivalent to $\bar{t}_1[\bar{t}_2/\bar{x}]$ by the equality of **PX**. On the other hand, from the induction hypotheses, we see $[\bar{t}_1[\bar{t}_2/\bar{x}], \tilde{t}_1[\tilde{t}_2/\tilde{x}]] : Eq_\tau$. By means of the following substitution lemma, $\widetilde{t_1[t_2/x]}$ is equivalent to $\tilde{t}_1[\tilde{t}_2/\tilde{x}]$ by the equality Eq_τ. These imply (3). (1) is obvious from the induction hypothesis and (2) is provable by the substitution lemma. □

The key of the above proof is the following substitution lemma.

Substitution lemma. *If $\Gamma, x : \sigma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \sigma$ are typings of PT and their interpretations are provable in* **PX**, *then the following is provable in* **PX**:

$$\hat{\Gamma} \Rightarrow [\bar{t}_1[\bar{t}_2/\bar{x}], \overline{t_1[t_2/x]}] : Eq_\tau .$$

This appears obvious, but it is not. For example, $\overline{(\lambda x.xy)[z/y]}$ is $\Lambda(\bar{z} = \bar{z})(\lambda(\bar{x})(app{*}(\bar{x}, \bar{z})))$, but $\overline{\lambda x.xy}[\bar{z}/\bar{y}]$ is $\Lambda(\bar{x} = \bar{z})(\lambda(\bar{x})(app{*}(\bar{x}, \bar{y})))$. They are not α-convertible in the sense of **PX**. Their values are different λ-closures, so we have to show they are extensionally equal as functions of an appropriate type $\sigma \to \tau$, respecting the equality of the type. The substitution lemma is proved by the induction on the complexity of t_1 by means of the following lemma.

Lemma

(A) *If $\Gamma \vdash x : \tau$ is provable, then there is a σ such that (i) $x : \sigma \in \Gamma$, (ii) σ is the form $\forall\alpha_1 \ldots \forall\alpha_n.\sigma_0$ (n may be 0) and τ is $\sigma_0[\sigma_0/\alpha_0] \ldots [\sigma_n/\alpha_n]$ for some $\sigma_1, \ldots, \sigma_n$.*

(B) *If $\Gamma \vdash t_1t_2 : \tau$ is provable, then τ is α-convertible to the form $\forall\alpha_1 \ldots \alpha_n.\sigma_1$ (n may be 0) such that $\alpha_1, \ldots, \alpha_n$ are not free variables of Γ, and the following are provable:*

$$\Gamma \vdash t_1 : \sigma_2 \to \sigma_1 \qquad \Gamma \vdash t_2 : \sigma_2$$

(C) *If $\Gamma \vdash \lambda y.t : \tau$ is provable, then τ is α-convertible to $\forall\alpha_1 \ldots \forall\alpha_n.(\sigma_1 \to \sigma_2)$ (n may be 0) such that $\alpha_1, \ldots, \alpha_n$ are not free variables of Γ, and the following is provable:*

$$\Gamma, y : \sigma_1 \vdash t : \sigma_2$$

This lemma can be proved by means of an induction on the length of derivation of typings. Proofs of these two lemmas are left for the reader.

The complication of the proof of the above theorem is due to by the interpretation of the λ-function. There is an easy way to avoid this. We can define call-by-value combinators k and s as follows:

$$k \equiv \Lambda(\lambda(x)\Lambda(\lambda(y)(x))),$$
$$s \equiv \Lambda(\lambda(x)\Lambda(\lambda(y)\Lambda(\lambda(z)(app{*}(app{*}(x, y), app{*}(x, z)))))).$$

By means of the translation of the λ-calculus to the call-by-value combinatory logic of s and k, we see a substitution lemma holds without any assumption of type assignments. The cost of this approach is that the translated **PX** expressions are more complicated and slower than the ones in the interpretation given above.

The semantics given above is essentially an impredicative version of the formalized semantics of Martin-Löf's type theory without type hierarchy of Beeson 1985. It is possible to extend type system so as including various other types, e.g., Martin-Löf's propositional equality $I(A, x, y)$, intersection types of Coppo and Dezani-Ciancaglini 1979, and even a subtype judgment such as $\tau_1 \subseteq \tau_2$.

6 Semantics

In this chapter, we give a mathematical semantics for **PX**. Although we pronounce **PX** to be a constructive theory, we give the semantics in the framework of classical mathematics. We *must* do so, for **PX** includes classical reasoning via the modal sign. We *will* do so, for we wish the extracted programs to meet specifications in the usual rather than the constructive sense (see 4.1).

The basic idea of the semantics is due to Feferman 1979. Feferman gave a number of semantics for his theories of functions and classes on which **PX** is based. The one we use is one of them. The semantics employs three stages. First, Feferman gives a semantics of an axiom system APP. This is done to give a semantics of the "programming language" used in his theories. Secondly, he gives a semantics of formulas relative to an arbitrary semantics of $Class(a)$ and $a_1 : a_2$. Lastly, he gives a semantics of classes via this relative semantics of formulas. Over the semantics of classes, a nonrelative semantics of formulas is automatically given by the relative semantics. Our semantics of **PX** takes these three steps, too. But there are differences between Feferman's and our semantics. In the second step, Feferman uses the usual semantics of the usual logic of total terms, but we use a semantics of logic of partial terms (LPT). In the third step, Feferman uses ordinals to construct a model of classes. We use a fixed point theorem instead. Our construction is essentially the same as Feferman's, but is more easily understandable.

The three steps will be presented in 6.1, 6.2, and 6.3, respectively. In 6.4, we will show that MGA is satisfied.

6.1. Semantics of computation

In this section, we give a semantics of the DEF-system. the DEF-system is a dialect of Lisp and our semantics is given by its interpreter. First, as in the case of Lisp, we define MS-translation, i.e., we define an injection from the DEF-systems to the set of S-expressions Obj. Let $\langle D, E, F\rangle$ be a DEF-system constructed from sets of variables, say VV, constants, say C, identifiers for basic functions, say B, and identifiers for other (user defined) functions, say FI. Since D, E, F are countable sets, there is an injection *code* such that

$$code : B \amalg C \amalg FI \amalg VV \amalg Reserved \to Latom,$$

where *Reserved* is the set of "reserved words" *cond*, Λ, λ, and *let*, and *Latom* is the set of literal atoms, i.e., the set $Atm - N$. We fix such an injection *code* in the rest of this section. For simplicity, we often denote $code(\alpha)$ by α^*. We extend the map *code* to expressions E, functions F, and definitions D as follows:

1. When e is an expression $f(e_1, \ldots, e_n)$, e^* is
$$(f^*\ {e_1}^* \ldots {e_n}^*).$$
2. When e is an expression $cond(e_1, d_1; \ldots, e_n; d_n)$, e^* is
$$(cond^*\ ({e_1}^*\ {d_1}^*) \ldots ({e_n}^*\ {d_n}^*)).$$
3. When e is an expression $\Lambda(v_1 = e_1, \ldots, v_n = e_n)(f)$, e^* is
$$(\Lambda^*\ (({v_1}^*\ {e_1}^*) \ldots ({v_n}^*\ {e_n}^*))\ f^*).$$
4. When f is a function $\lambda(v_1, \ldots, v_n)(e)$, f^* is
$$(\lambda^*\ ({v_1}^* \ldots {v_n}^*)\ e^*).$$
5. When e is an expression $let\ p_1 = e_1, \ldots, p_n = e_n\ in\ e$, e^* is
$$(let^*\ (({p_1}^*\ {e_1}^*) \ldots ({p_n}^*\ {e_n}^*))\ e^*).$$
6. When d is a definition $\{f_1 \vec{v}_1 = e_1, \ldots, f_n \vec{v}_n = e_n\}$, d^* is
$$(({f_1}^*(\vec{v}_1^*)\ {e_1}^*) \ldots ({f_n}^*(\vec{v}_n^*)\ {e_n}^*)).$$

The interpreter of the DEF-system consists three evaluators, *Eval*, *Func*, and *Def*, which compute the value of expressions, functions, and definitions, respectively. Nontheless, for mathematical exactness and clarity and also for technical reasons related to the proof of the validity of MGA, we will give two different denotations of the interpreter according to the method of Plotkin 1985.

First, we will give an overview of Plotkin's denotational semantics. Plotkin 1985 showed how to rephrase the D_∞-construction in his setting, but this is not necessary for our purpose, so we will not pursue it. The basic point of Plotkin's new approach to Scott theory is the change of the definition of cpo.

Definition 1 (cpo in Plotkin's sense). A *cpo in Plotkin's sense* is a partially ordered set satisfying the condition: any ascending chain $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ has a sup $\bigsqcup a_n$.

Note that we do not assume the existence of a bottom element. So each plane set S can be considered to be a cpo without adding a bottom element. Such a cpo is called a *flat domain*. The finite or infinite product of cpo's is a cpo by coordinatewise ordering, e.g.,

$$\langle a_1, a_2 \rangle \sqsubseteq \langle b_1, b_2 \rangle \text{ iff } a_i \sqsubseteq b_i \ (i = 1, 2).$$

The projection operator to a component D_λ from the product $\prod_{\lambda \in \Lambda} D_\lambda$ is denoted by proj_λ. The finite or infinite coproduct of cpo's is also a cpo. Note that it is not necessary to add a new bottom. The order of the coproduct $\coprod_{\lambda \in \Lambda} D_\lambda$ is given by

$$a \sqsubseteq b \text{ iff } a \text{ and } b \text{ are in the same } D_\lambda \text{ and } a \sqsubseteq b \text{ in } D_\lambda.$$

The injection operator from D_λ to $\coprod_{\lambda \in \Lambda} D_\lambda$ is denoted by inj_λ.

In Scott's theory of computation, a partial function is identified with the total function, which takes a bottom when it is not defined for an input. Since a cpo in Plotkin's sense does not necessarily have a bottom element, we cannot follow this convention. So we use partial functions rather than total functions.

Definition 2 (partial function). A *partial function* from A to B is a mapping from A to B that is not necessarily defined for all elements of A, i.e., its *graph* is a set $G \subseteq A \times B$ that satisfies the condition: if $\langle x, y \rangle \in G$ and $\langle x, z \rangle \in G$, then $y = z$. We will indicate by $\mathit{graph}(f)$ the graph of f. (From the set-theoretical standpoint f should be identified with $\mathit{graph}(f)$.) When f is a partial function from A to B, we write $f : A \rightharpoonup B$.

In Scott's theory, there is a difficulty in the interpretation of bottom. In his theory, a bottom is not always an "undefined value", e.g., the bottom element of a function space is not an undefined value but a value of the function which is undefined for every input. There is no simple criteria by which we can decide if a bottom stands for an undefined value or a meaningful value. Since in Plotkin's theory no actual value stands for an undefined value, this problem does not appear. This is good, but it causes another problem. In Scott's theory every expression is assumed to have a value, but this is not so in Plotkin's theory, where partial functions are used. We have to introduce new notational conventions by which we may denote undefined expressions. The notational conventions we are going to introduce are essentially the same as those of the logic of partial terms (LPT) of **PX**. But it should be noted this system of notational conventions is meta-level. Even though our informal arguments look like arguments in LPT of **PX**, we have to distinguish them from the formalized arguments of LPT of **PX**, since we will give a semantics of LPT of **PX** by them.

First, we have to define when an expression has a value for each expression we use. It is sufficient to define an expression's *definedness* and *value* for each notation which builds a compound expression from other expressions. We will use α, β, $\gamma, \ldots$ as variables for expressions. These appear to be similar to partial variables of **PX**, but they are different. We may say "let α be an expression that has the form $if\ \beta\ then\ \gamma$", but we cannot talk about the syntactic form of a value of partial variable. On the contrary, a, b, c, u, v, x, y, z with or without indices stand for actual values. These just correspond to total variables of **PX**. The most fundamental expression that may fail to have a value is a *function application* $f(\alpha)$. A partial function f is said to be defined for an input x, when there is an element y for which $\langle x, y\rangle$ belongs to the graph of f, and the value of $f(x)$ is y such that $\langle x, y\rangle \in graph(f)$. But this is not sufficient, we have defined the definedness of the expression $f(x)$ only when x is an actual value. The definedness and value of a general function application $f(\alpha)$ is defined as follows: $f(\alpha)$ is defined iff the expression α is defined and has a value x, and $f(x)$ is defined, and the value of $f(\alpha)$ is defined to be the value of $f(x)$. Namely, function application is computed by the call-by-value rule.

When an expression α has a value, we will write $E(\alpha)$ and denote its value by the expression α itself.

As opposed to the usual mathematical language and so to the usual computational languages, a conditional expression must be distinguished from partial functions, since it does not obey the call-by-value rule. The expression "$if\ \alpha\ then\ \beta\ else\ \gamma$" is defined as follows: if α has the value $\boldsymbol{true}$, and β is defined, then it is defined and its value is β, and if α has the value $\boldsymbol{false}$ and γ is defined, then it is defined and its value is γ.

An *abstraction* $\lambda x \in A.\alpha$ stands for the partial function whose graph is

$$\{\langle x, y\rangle | x \in A, \alpha \text{ is defined and its value is } y\}.$$

Note that an abstraction is always defined and denotes this value.

The equation $e_1 \simeq e_2$ means if e_1 has a value then e_2 also has the same value and vice versa. This corresponds to the equality of **PX**. The usual equality sign $\alpha = \beta$ is used for an abbreviation for $E(\alpha)$ & $E(\beta)$ & $\alpha \simeq \beta$. The inequality $\alpha \underset{\sim}{\sqsubset} \beta$ means if α has a value then β also has a value and the value of α is equal to or smaller than the value of β. On the contrary, $\alpha \sqsubseteq \beta$ means that α and β are defined and the value of α is smaller than or equal to the value of β. Note that $\alpha \simeq \beta$ is equivalent to $\alpha \underset{\sim}{\sqsubset} \beta$ & $\beta \underset{\sim}{\sqsubset} \alpha$. Assume $\alpha_0 \underset{\sim}{\sqsubset} \alpha_1 \underset{\sim}{\sqsubset} \ldots$. Then the expression $\bigsqcup \alpha_m$ is defined iff there is an m such that α_n is defined. If such an m exists, e_n has a value, say a_n, for all $n \geq m$, then the value of $\bigsqcup \alpha_n$ is $\bigsqcup_{n \geq m} a_n$. In the following $undefined$ is an any fixed undefined *expression*, and $\alpha \neq \beta$ means α and β are defined and have different values.

This provides enough notational conventions to define the interpreter of a DEF-system, except for recursive definitions, for which we need the concept of continuity.

Definition 3 (partial continuous function). A *partial continuous function* f from a cpo D_1 to a cpo D_2 is a partial function from D_1 to D_2 satisfies the conditions:

(i) f preserves $\precsim$, i.e., $\forall x, y \in D_1 . x \precsim y \supset f(x) \precsim f(y)$.
(ii) if $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$, then $f(\bigsqcup a_n) \simeq \bigsqcup f(a_n)$.

We will write $[D_1 \rightharpoonup D_2]$ for the set of continuous partial function from D_1 to D_2. The partial order of cpo of partial continuous functions is defined by

$$f_1 \sqsubseteq f_2 \text{ iff } \forall x \in D_1 . f_1(x) \precsim f_2(x).$$

The totally undefined function $\bot$ defined by $\forall x . \bot(x) = \bot$ is the bottom element of the cpo. If $\lambda f \in [D_1 \rightharpoonup D_2] . \alpha$ defines a continuous *total* automorphism of $[D_1 \rightharpoonup D_2]$, say Φ, then the partial continuous function f that is defined by the recursive definition $f = \alpha$ is the minimal fixed point $fix(\Phi)$, i.e., $\bigsqcup \Phi^n(\bot)$. Simultaneous recursive definition is defined similarly. A partial function with n arguments $f(x_1, \ldots, x_n)$ is continuous iff f is continuous for each argument x_i. Projections of products and injections of sums are all continuous. Note that $proj_i(x)$ is continuous with respect to not only x but also the index i. We consider the set of indices to be a flat domain.

We have introduced four constructors of expressions: application, abstraction, conditional expression, and recursive definition. When an expression α is built up from variables and constants of cpo's through these constructors, it is continuous with respect to each variable x, i.e., $\lambda x . \alpha$ is continuous. We recognize that all function variables run over some spaces of partial *continuous* functions. Strictly speaking, we are working in the category of cpo's and partial continuous functions.

Even though the notational conventions of informal LPT are sufficient, in order to define the interpreter of the DEF-system we need a concept of environment. In the following, we use the notation

$$[x \in A \rightarrow B(x)] = \{f \in [A \rightarrow \coprod_{x \in A} B(x)] | \forall x \in A . f(x) \in B(x)\},$$

$$[x \in A \rightharpoonup B(x)] = \{f \in [A \rightharpoonup \coprod_{x \in A} B(x)] | \forall x \in A . E(f(x)) \supset f(x) \in B(x)\},$$

$$x \in A \times B(x) = \{\langle x, y \rangle | x \in A \wedge y \in B(x)\},$$

$$Hom(x) = \begin{cases} [Obj^i \rightharpoonup Obj] & \text{if } x \in N \\ \prod_{i \in x} [Obj^i \rightharpoonup Obj] & \text{otherwise.} \end{cases}$$

If A is a cpo and $B(x)$ is a cpo for each $x \in A$, then $[x \in A \rightarrow B(x)]$, $[x \in A \rightharpoonup B(x)]$, $x \in A \times B(x)$ are canonically cpo's.

Definition 4 (environment). A *constant environment* is a *total* function γ from C^* to *Obj* satisfying the condition: (i) $\gamma(0)=0$, $\gamma(quote(\alpha))=\alpha$, (ii) $\gamma(t)$, $\gamma(nil)$, $\gamma(V)$, $\gamma(N)$, $\gamma(Atm)$, $\gamma(T)$ are literal atoms different from each other. We denote the set of constant environments by Env_C. A *value environment* is a *partial* function ρ from VV^* to *Obj* satisfying the condition: ρ is defined for each total variable of VV_t. We denote the set of value environments by Env_V. A *function environment* ρ is a *total* function of $[f \in FI^* \to Hom(arity(f))]$. Recall $arity(f)$ is the arity of f. A *basic function environment* β is a partial function of $[f \in (B - \{app, app*\})^* \rightharpoonup Hom(arity(f))]$ satisfying: (i) each basic function identifier must be mapped to its intended meaning, e.g., $\beta(suc)$ is the successor function over natural numbers, (ii) $proj_n(\beta(list))$ is the n-argument list function. We will write Env_B and Env_F for the sets of basic function environments and function environments, respectively.

Note that a basic function environment does *not* give a denotation of *app*, or *app*∗, but a function environment does.

Let ρ be a value environment and let $x_1, \ldots, x_n \in VV^*$ and $v_1, \ldots, v_n \in Obj$. Then a new environment changing the values of x_i $\rho[v_0/x_0, \ldots, v_n/x_n]$ is defined by

$$\rho[v_0/x_0, \ldots, v_n/x_n](x) = \begin{cases} v_i & \text{if } x \text{ is } x_i, \\ \rho(x) & \text{otherwise.} \end{cases}$$

We will denote $\perp[\ldots]$ by $[\ldots]$ and $\rho[a_1/b_1, \ldots, a_n/b_n]$ by $\rho[a_1, \ldots, a_n/b_1, \ldots, b_n]$. Strictly speaking, we defined a *total continuous* function *change*

$$change \in [Env_V \times (a \in List(VV^*) \times List_{length(a)}(Obj)) \to Env_V],$$

such that

$$change(\rho, (x_0, \ldots, x_n), (v_0, \ldots, v_n)) = \rho[v_0/x_0, \ldots, v_n/x_n],$$

where $List(VV^*)$ is the cpo of the list of VV^* and $List_{length(a)}(Obj)$ is the cpo of lists of objects whose length is the same as the length of the list a. Hence if one of $\alpha_1, \ldots, \alpha_n$ is undefined, then $\rho[\alpha_1/x_1, \ldots, \alpha_n/x_n]$ is undefined. The operation $\rho[a_0/b_0, \ldots, a_n/b_n]$ is continuous with respect to ρ, a_i and b_i.

Now we give the first definition of the *interpreter* for the DEF-system. The reason why we present the definition is to show what is going on from a mathematical point of view. The point is that meanings of basic functions except *app*∗, *app* are fixed, but the meanings of *app* and *app*∗ must be determined through the definitions of user-defined functions. Since this makes much use of function spaces, it is not immediately clear that the interpreter is actually implementable on machines. We later give another definition, which is essentially a Lisp evaluator, and see the equivalence of the two definitions.

A DEF-system is a system of function definitions, expressions, and function expressions, so their denotations consist of a function environment, say *Def*, a value of an expression e under a value environment ρ, say $\mathit{Value}(e,\rho)$, and a function of a function expression f under a value environment ρ, say $\mathit{Func}(f,\rho)$. Namely, *Def*, *Value*, and *Func* have the types

$$\begin{aligned}
&\mathit{Def} \in \mathit{Env}_F,\\
&\mathit{Value} \in [E^* \times \mathit{Env}_V \rightharpoonup \mathit{Obj}],\\
&\mathit{Func} \in [x \in (F^* \times \mathit{Env}_V) \to \mathit{Hom}(\mathit{arity}(\mathit{code}^{-1}(\mathit{proj}_1(x))))].
\end{aligned}$$

The denotations of the basic functions except *app*, *app*∗ are given by the basic function environment. On the other hand, the denotations of *app* and *app*∗ are simultaneously defined with *Def*, *Value*, and *Func,* for this depends on the function definitions. We denote the denotations of *app* and *app*∗ by *App*, *App*∗, respectively. These two denotations have the types

$$\mathit{App} \in \mathit{Hom}(2), \quad \mathit{App}{*} \in \mathit{Hom}(N^+).$$

We will give simultaneous equations below whose minimal solution is these denotations. Strictly speaking, the equations have parameters γ, constant environment, and β, a basic function environment, so the solution depends on these. But they are fixed, so we omit such parameters. For simplicity, we restrict patterns of *let* to variables.

In the definition, we use the following conventions. We recognize a predicate $P(a)$ on a flat domain as a *total* continuous function f to the domain $\{\boldsymbol{true}, \boldsymbol{false}\}$ such that $f(a) = \mathit{true}$ iff $P(a)$ holds. The membership relation $\in$ and its negation $\notin$ are regarded as binary predicates. So $e_1 \notin e_2$ implies e_1 and e_2 are defined. FC is the set of function closures, i.e., the S-expressions of the form f^*, where f is in FI or a closed function expression of F of the form

$$\lambda\ (a_1 \ldots a_n)\ (\mathit{let}\ b_1 = \mathit{quote}(\alpha_1), \ldots, b_m = \mathit{quote}(\alpha_m)\ \mathit{in}\ f),$$

where $\alpha_1, \ldots, \alpha_m$ are arbitrary S-expressions. $\mathit{Arity}(a)$ is defined by

$$\mathit{Arity}(a) = \begin{cases} \{\mathit{arity}(\mathit{code}^{-1}(a))\} & \text{if } \mathit{arity}(\mathit{code}^{-1}(a)) \in N,\\ \mathit{arity}(\mathit{code}^{-1}(a)) & \text{otherwise.}\end{cases}$$

List is the set of lists. If $f \in FI^*$ then $\mathit{def}(f)$ and $\mathit{vars}(f)$ are the function body and formal parameters of the definition of f, respectively. If f is in $\mathit{Hom}(n)$ ($n \in N$), then $\mathit{fetch}(n, f)$ is f and if f is in $\mathit{Hom}(P)$ ($P \subseteq N$) and $n \in P$, then $\mathit{fetch}(n, f)$ is $\mathit{proj}_n(f)$. The variables a, b, c, ... and these with indexes run over

Obj and when we write, e.g., $a \equiv (b\ c)$, it means the S-expressions a is a list of b and c. $l(a)$ is the length of a list a and $nth(i, a)$ is the ith element of a.

```
$Value(a,\rho) \simeq$
 $if\ a \notin E^*\ then\ undefined$
  $else\ if\ a \in C^*\ then\ \gamma(a)$
  $else\ if\ a \in VV^*\ then\ \rho(a)$
  $else\ if\ a \equiv (cond^*\ (a_1\ b_1)\ldots(a_n\ b_n))\ then$
        $if\ n = 0\ then\ nil^*$
          $else\ if\ Value(a_1,\rho) \not\equiv nil^*\ then\ Value(b_1,\rho)$
            $else\ Value((cond^*\ (a_2\ b_2)\ldots(a_n\ b_n))),\rho)$
  $else\ if\ a \equiv (\Lambda^*\ ((a_1\ b_1)\ldots(a_n\ b_n))\ f)\ then$
        $if\ f \equiv (\lambda^*(c_1\ldots c_m)\ e)\ then$
          $(\lambda^*\ (c_1\ldots c_m)$
               $(let^*\ ((a_1\ (quote^*\ Value(b_1,\rho)))\ldots(a_n\ (quote^*\ Value(b_n,\rho))))\ e))$
        $else\ f$
  $else\ if\ a \equiv (let^*\ ((a_1\ b_1)\ldots(a_n\ b_n))\ c)\ then$
        $Value(c,\rho[Value(b_1,\rho)/a_1,\ldots,Value(b_n,\rho)/a_n])$
  $else\ if\ a \equiv (f\ a_1\ldots a_n)\ then$
        $fetch(n, Func(f,\rho))(Value(a_1,\rho),\ldots,Value(a_n,\rho))$
  $else\ undefined$

$Func(f,\rho) \simeq if\ f \notin F^*\ then\ \lambda a.undefined$
               $else\ if\ f \in B^*\ then\ \beta(f)$
               $else\ if\ f \in FI^*\ then\ Def(f)$
               $else\ if\ f \equiv app^*\ then\ App$
               $else\ if\ f \equiv app{*}^*\ then\ App{*}$
               $else\ if\ f \equiv (\lambda^*\ (a_1\ \ldots\ a_m)\ b)\ then$
                    $\lambda x_1,\ldots,x_m.Value(b,\rho[x_1/a_1,\ldots,x_m/a_m])$
               $else\ \lambda a.undefined$
```

$$App \simeq \lambda a, b.if\ a \notin FC\ or\ b \notin List\ or\ l(b) \notin Arity(a)\ then\ undefined$$
$$else\ fetch(l(b), Func(a, \bot))(nth(1, b), \ldots, nth(l(b), b))$$

$$App{*} \simeq \lambda l \in N^+.\lambda a, b_1, \ldots, b_{l-1}.if\ a \notin FC\ or\ l-1 \notin Arity(a)\ then\ undefined$$
$$else\ fetch(l, Func(a, \bot))(b_1, \ldots, b_{l-1})$$

$$Def(a) \simeq \lambda b_1, \ldots, b_{arity(code^{-1}(a))}.Value(def(a), [b_1, \ldots, b_{arity(code^{-1}(a))}/vars(a)])$$

We used the notation $\rho[e_1/x_1, \ldots, e_n/x_n]$ in the above equations. Strictly speaking, those must be expressed by the function *change* mentioned above. We do not explain how to do that, since it is easy.

The right hand sides of these equations are built up from variables and constants of cpo's by means of application, abstraction, and conditional expression. So they are continuous with respects to free variables, and the minimal solution of the equations exists.

Next we give another description of *Value* which is essentially a Lisp interpreter. The point of the new interpreter is that it does not give a denotation of a function, so it does not involve the function spaces. We define two evaluators *Apply* and *Eval* by means of the following equations:

```
$Apply(a, b, \rho) \simeq$
  $if\ a \notin FC\ or\ b \notin List\ or\ l(b) \notin Arity(a)\ then\ undefined$
    $else\ if\ a \in FI^*\ then\ Eval(def(a), [nth(1, b)/v_1^a, \ldots, nth(l(b), b)/v_{l(b)}^a])$
    $else\ if\ a \equiv (\lambda^*\ (v_1 \ldots v_n)\ c)\ then$
      $Eval(c, \rho[nth(1, b)/v_1, \ldots, nth(l(b), b)/v_{l(b)}])$
    $else\ if\ a \equiv app^*\ then$
          $if\ l(b) \not\equiv 2\ or\ nth(1, b) \notin FC\ then\ undefined$
            $else\ Apply(nth(1, b), nth(2, b), \bot)$
    $else\ if\ a \equiv app{*}^*\ then$
          $if\ nth(1, b) \notin FC\ or\ l(b) - 1 \notin Arity(nth(1, b))\ then\ undefined$
                $else\ Apply(nth(1, b), tail(b), \bot),$
```

where $tail(b)$ is the tail of the list b, and $v_1^a, \ldots, v_{l(b)}^a$ is $vars(a)$.

```
$Eval(a, \rho) \simeq$
  $if\ a \notin E^*\ then\ undefined$
    $else\ if\ a \in C^*\ then\ \gamma(a)$
    $else\ if\ a \in VV^*\ then\ \rho(a)$
    $else\ if\ a \equiv (cond^*\ (a_1\ b_1) \ldots (a_n\ b_n))\ then$
          $if\ n = 0\ then\ nil^*$
            $else\ if\ Eval(a_1, \rho) \not\equiv nil^*\ then\ Eval(b_1, \rho)$
              $else\ Eval((cond^*\ (a_2\ b_2) \ldots (a_n\ b_n))), \rho)$
    $else\ if\ a \equiv (\Lambda^*\ ((a_1\ b_1) \ldots (a_n\ b_n))\ f)\ then$
          $if\ f \equiv (\lambda^*(c_1 \ldots c_m)\ e)\ then$
            $(\lambda^*\ (c_1 \ldots c_m)$
                $(let^*\ ((a_1\ (quote^*\ Eval(b_1, \rho))) \ldots (a_n\ (quote^*\ Eval(b_n, \rho))))\ e))$
          $else\ f$
    $else\ if\ a \equiv (let^*\ ((a_1\ b_1) \ldots (a_n\ b_n))\ c)\ then$
          $Eval(c, \rho[Eval(b_1, \rho)/a_1, \ldots, Eval(b_n, \rho)/a_n])$
    $else\ if\ a \equiv (f\ a_1 \ldots a_n)\ then$
          $Apply(f, (Eval(a_1, \rho), \ldots, Eval(a_n, \rho)), \rho)$
    $else\ undefined$
```

Then $Eval \sqsubseteq Value$ holds, for if we set

$$Eval = Value, \quad Apply = \lambda a, b, \rho.\, Value((app^*\ x\ y), \rho[a/x, (quote^*\ b)/y]),$$

then the definition equations of *Eval* and *Apply* hold. In the same way, we can prove that $Value \sqsubseteq Eval$. Hence *Eval* equals *Value*.

Theorem 1. *The two descriptions of* **PX** *interpreter are equivalent.*

It would be clear that if a Lisp interpreter evaluates an expression of **PX** to a value, then *Eval* evaluates it to the same value as far as Λ is implemented as an appropriate macro. But the reverse is not true, for the usual Lisp interpreter does not check the first argument of *apply* to be in FC.

Claim. When an expression has a value v in the semantics of **PX**, then it is evaluated to the same value by the ordinary Lisp interpreter.

This maintains that we may evaluate extracted programs by Lisp. The interpreter enjoys the usual semantic properties of formal languages, e.g., the substitution lemma. Set

$$\widetilde{Value}(e,\rho) = Value(e^*,\rho), \quad \widetilde{Func}(f,\rho) = Func(f^*,\rho).$$

Then the substitution lemma is stated as follows:

Substitution lemma for $\widetilde{Value}$. *If e_1 and e_2 are expressions, f is a function ,and x is a free variable of e_1, then the following hold:*

$$\widetilde{Value}(e_1[e_2/x],\rho) \simeq \widetilde{Value}(e_1,\rho\langle \widetilde{Value}(e_2,\rho)/x\rangle),$$

$$\widetilde{Func}(f[e/x],\rho) \simeq \widetilde{Func}(f,\rho\langle \widetilde{Func}(e,\rho)/x\rangle),$$

where $\rho\langle\alpha/x\rangle$ is defined by

$$\rho\langle\alpha/x\rangle(y) = \begin{cases} \alpha & \textit{if } x \textit{ is } y, \\ \rho(x) & \textit{otherwise.} \end{cases}$$

Note that $\rho\langle\alpha/x\rangle$ is always defined.

This can be proved by the induction on the complexity of e_1 and f. It is also easy to see that $\widetilde{Value}(e,\rho_1) \simeq \widetilde{Value}(e,\rho)$, whenever ρ_1 and ρ_2 coincide on the free variables of e. These properties may fail for a Lisp interpreter, even if its scoping rule is lexical. The substitution property strongly depends on the variable condition and execution rules on Λ-expressions.

6.2. Tarskian semantics of formulas

In this section, we give the semantics of formulas relative to an arbitrary interpretation of classes. Note that we give the semantics using classical logic. This is inevitable, for **PX** includes the axioms on $\diamondsuit$. Each of the soundness theorems in this and the next section gives a soundness result not only for the theory stated there but also for the theory enhanced with *classical logic*.

Let **Cl** be an arbitrary subset of *Obj* and let **Ext** be an arbitrary *total* function from **Cl** to the power set of *Obj*, *P*(*Obj*) in notation. The intended meaning of **Cl** is the set of classes and **Ext**(c) is intended to be the extension of c, i.e., $\{x|x : c\}$. Then we define Tarskian semantics of formulas relative to **Cl** and **Ext**. The truth value of the formula of F is denoted by $[\![F]\!]_\rho^{\mathbf{Cl},\mathbf{Ext}}$. The conditional formula

$$e_1 \to F_1; \ldots ; e_n \to F_n$$

is recognized as an abbreviation of

$$(e_1 : T \wedge F_1) \vee (e_2 : T \wedge e_1 = nil \wedge F_2) \vee \ldots \vee (e_n : T \wedge e_{n-1} = nil \wedge \ldots \wedge e_1 = nil \wedge F_n),$$

and

$$\nabla p_1 = e_1, \ldots, x_n = e_n.F,$$

is recognized as an abbreviation of

$$\exists \vec{x}.(exp(p_1) = e_1 \wedge \ldots \wedge exp(p_n) = e_n \wedge F),$$

where $\vec{x}$ is the sequence of the variables $FV(p_1) \cup \ldots \cup FV(p_n)$. So we do not give semantics of these formulas.

In the definition of $[\![F]\!]^{\mathbf{Cl,Ext}}_{\rho}$, we will use the following conventions. The notation $\alpha \in \beta$ means α and β have values v_1 and v_2 such that v_2 is a set to which v_1 belongs. We will use the tuple notation $[\alpha_1, \ldots, \alpha_n]$ as a macro just as explained in 2.2. Then the Tarskian semantics of formulas is given by

$$[\![E(e)]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} E(\widetilde{Value}(e, \rho))$$

$$[\![Class(e)]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} \widetilde{Value}(e, \rho) \in \mathbf{Cl}$$

$$[\![e_1 = e_2]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} \widetilde{Value}(e_1, \rho) \simeq \widetilde{Value}(e_2, \rho)$$

$$[\![[e_1, \ldots, e_n] : e]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=}$$

$$\widetilde{Value}(e) \in \mathbf{Cl} \wedge [\,\widetilde{Value}(e_1, \rho), \ldots, \widetilde{Value}(e_n, \rho)] \in \mathbf{Ext}(\widetilde{Value}(e, \rho))$$

$$[\![\top]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} \boldsymbol{true}$$

$$[\![\bot]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} \boldsymbol{false}$$

$$[\![F_1 \wedge \ldots \wedge F_n]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} [\![F_1]\!]^{\mathbf{Cl,Ext}}_{\rho} \wedge \ldots \wedge [\![F_n]\!]^{\mathbf{Cl,Ext}}_{\rho}$$

$$[\![F_1 \vee \ldots \vee F_n]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} [\![F_1]\!]^{\mathbf{Cl,Ext}}_{\rho} \vee \ldots \vee [\![F_n]\!]^{\mathbf{Cl,Ext}}_{\rho}$$

$$[\![F_1 \supset F_2]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} [\![F_1]\!]^{\mathbf{Cl,Ext}}_{\rho} \supset [\![F_2]\!]^{\mathbf{Cl,Ext}}_{\rho}$$

$$[\![\neg F]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} \neg [\![F]\!]^{\mathbf{Cl,Ext}}_{\rho}$$

$$[\![\Diamond F]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=} [\![F]\!]^{\mathbf{Cl,Ext}}_{\rho}$$

$$[\![\forall \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n.F]\!]^{\mathbf{Cl,Ext}}_{\rho} \overset{\text{def}}{=}$$

$$\bigwedge_{i=1,\ldots,n} \widetilde{Value}(e_i, \rho) \in \mathbf{Cl} \;\supset \forall \vec{v}_1, \ldots, \vec{v}_n.(R_i(\vec{v}) \supset [\![F]\!]^{\mathbf{Cl,Ext}}_{\rho[\vec{v}_1/\vec{x}_1, \ldots, \vec{v}_n/\vec{x}_n]})$$

$$[\![\exists \vec{x}_1 : e_1, \ldots, \vec{x}_n : e_n . F]\!]_\rho^{\mathbf{Cl},\mathbf{Ext}} \stackrel{\mathrm{def}}{=}$$
$$\bigwedge_{i=1,\ldots,n} \widetilde{Value}(e_i, \rho) \in \mathbf{Cl} \ \wedge \exists \vec{v}_1, \ldots, \vec{v}_n . (R_i(\vec{v}) \wedge [\![F]\!]^{\mathbf{Cl},\mathbf{Ext}}_{\rho[\vec{v}_1/\vec{x}_1, \ldots, \vec{v}_n/\vec{x}_n]}),$$

where $R_i(\vec{v})$ is the formula defined by

$$\bigwedge_{i=1,\ldots,n} \vec{v}_i \in \mathbf{Ext}(\widetilde{Value}(e_i, \rho)) \wedge \bigwedge_{j=1,\ldots,m_i} v^i_j \in \mathbf{Cl},$$

so that $v^i_1, \ldots, v^i_{m_i}$ are the class variables among $\vec{v}_i$. Note that this definition is by induction on the complexity of formulas.

We say a value environment ρ satisfies *the class condition* iff $\rho(x)$ is in **Cl**, whenever x is a class constant or class variable. A formula F is said to be *valid relative to* **Cl** *and* **Ext**, when $[\![F]\!]_\rho^{\mathbf{Cl},\mathbf{Ext}}$ holds for all value environments satisfying the class condition. Then we have the following theorem:

Relative soundness theorem. *All the axioms and rules of section 2.3 except* ($char$), ($N1$)-($N3$), ($V1$), ($V3$), *and* ($V4$) *are valid relative to any* **Cl** *and* **Ext**.

The most difficult rules to verify are $(= 4)$, $(\forall E)$, $(\exists I)$, $(\nabla\forall E)$, $(\nabla\exists I)$ and ($inst$). These are verified by means of the following lemma:

Substitution lemma for $[\![F]\!]_\rho^{\mathbf{Cl},\mathbf{Ext}}$. $[\![F[e/x]]\!]_\rho^{\mathbf{Cl},\mathbf{Ext}}$ *holds iff* $[\![F]\!]_{\rho_1}^{\mathbf{Cl},\mathbf{Ext}}$ *holds, where* ρ_1 *is* $\rho\langle \widetilde{Value}(e, \rho)/x \rangle$.

In the next section, we will construct **Cl** and **Ext** that satisfy the remaining axioms and rules. The following lemma will be of use:

Lemma 1. *Let F be a CIG template. Then if the following conditions holds for each class constant and free class variable of F, say x,*

$$\widetilde{Value}(x, \rho) \in \mathbf{Cl}_1 \cap \mathbf{Cl}_2 \quad \textit{and} \quad \mathbf{Ext}_1(\widetilde{Value}(x, \rho)) = \mathbf{Ext}_2(\widetilde{Value}(x, \rho)),$$

then $[\![F]\!]_\rho^{\mathbf{Cl}_1,\mathbf{Ext}_1}$ *is equivalent to* $[\![F]\!]_\rho^{\mathbf{Cl}_2,\mathbf{Ext}_2}$.

Proof. Since F is a CIG template, e is one of the free class variables $\vec{X}$ in all of the form $[e_1, \ldots, e_n] : e$ appearing in F. Since there is no occurrence of the form $Class(e)$ and bound class variable in F, **Cl** and **Ext** affect the interpretation of F only through the interpretation of the right hand side of the form by **Ext**. Hence the conclusion is obvious by the condition of the lemma. □

6.3. Semantics of classes

In this section, we construct **Cl** and **Ext** so that the relative semantics of the previous section satisfies the axioms for class. The general construction is a simplification of the constructions due to Feferman 1979. If it is not necessary to validate the axioms of join and product the semantics is considerably simplified, and we may assume the set of classes to be a recursive set. So we first give a semantics of classes without join and product, and later give the full semantics. It should be noted that the condition "a CIG template is of rank 0" is meaningless for the semantics of this section, since, as was noted before, our semantics is given in the sense of classical model theory, so any formula F is equivalent to $\Diamond F$.

6.3.1. The restricted model

The idea of the construction is simple: we think of a class as a code of the definition by which its extension is defined. By the condition of CIG templates it is possible to know the extension of a class from the extensions of classes from which the class is defined. The key point is that CIG definition is a *predicative* definition of sets. As the definition of CIG templates, we will refer to definition 2 of 2.4. First we define the set of classes. Let Φ be the set of CIG templates $\langle \vec{a}, \vec{v}, \vec{X}, A \rangle$ as in 2.4.1. Recall that there is an injection *cigname* from Φ into the set of basic function names. We denote the function *code* $\circ$ *cigname* by *cigcode*. We assume the basic function environment β has the following additional conditions:

(i) For each CIG template τ, $\beta(\mathit{cigcode}(\tau))$ is a total one-to-one function whose range is disjoint to the image of the set of class constants by *code*.
(ii) If $\tau_1 \not\equiv \tau_2$ then, the ranges of two functions $\beta(\mathit{cigcode}(\tau_1))$ and $\beta(\mathit{cigcode}(\tau_2))$ are disjoint.

For example,

$$\mathit{cigcode}(\langle \vec{a}, \vec{v}, \vec{X}, A \rangle) \longmapsto \lambda \vec{v}.\mathit{list}((\mathit{quote}\ \mathit{cigcode}(\langle \vec{a}, \vec{v}, \vec{X}, A \rangle)^*), \vec{v})$$

satisfies these conditions. We denote the function $\beta(\mathit{cigcode}(\tau))$ by cig_τ for each CIG template τ. Then the set of classes $\mathcal{C}$ is defined by the following clauses:

(1) If c is a class constant, then c^* belongs to $\mathcal{C}$.
(2) Let τ be a CIG template $\langle \vec{a}, \vec{v}, \vec{X}, A \rangle$ and $\vec{v}$ be $v_1, \ldots, v_n$. Then

$$\mathit{cig}_\tau(\rho(v_1), \ldots, \rho(v_n))$$

belongs to $\mathcal{C}$ for any value assignment ρ with the class condition, i.e., $\rho(X_i)$ has been in $\mathcal{C}$ for any X_i of $\vec{X}$.

If the coding functions of cig_τ are appropriately chosen, then $\mathcal{C}$ is a recursive set, i.e., we can decide if an S-expression is a class. Next we define the extension $\mathcal{E}(c)$ for each class $c \in \mathcal{C}$ by induction on the construction of c:

(a) If c is V^*, then $\mathcal{E}(c)$ is Obj.
(b) If c is N^*, then $\mathcal{E}(c)$ is N.

(c) If c is Atm^*, then $\mathcal{E}(c)$ is $Atom$.
(d) If c is T^*, then $\mathcal{E}(c)$ is $\{x|x \neq nil\}$.
(e) If c is $cig_\tau(\rho(v_1), \ldots, \rho(v_n))$ and $\tau = \langle \vec{a}, \vec{v}, \vec{X}, A\rangle$, then $\mathcal{E}(c)$ is the minimal solution of the set equation

$$X = \{\vec{x}|[\![A]\!]^{\mathcal{C},\mathcal{E}_{X,\rho}}_{\rho[\vec{x}/\vec{a},c/X_0]}\},$$

where $\vec{X}$ is $X_0, \ldots, X_n$ and $\mathcal{E}_X$ is the function defined by

$$\mathcal{E}_{X,\rho}(x) = \begin{cases} X & \text{if } x = c \\ \mathcal{E}(x) & \text{if } x \in \{\rho(X_1), \ldots, \rho(X_n)\} \\ \text{arbitrary subset of } Obj & \text{otherwise.} \end{cases}$$

(f) Otherwise, $Ext(c)$ is an arbitrary subset of Obj.

Then the following theorem holds:

The restricted soundness theorem. *All of axioms and rules of* **PX** *except* ($Product$) *and* ($Join$) *are valid under the semantics relative to* $\mathcal{C}$ *and* $\mathcal{E}$. *Furthermore, the axiom of the decidability of* $\mathcal{C}$ *is valid, too, i.e., there is an individual constant* cl *for which the following is an axiom:*

$$\forall x.\nabla y = app*(cl, x).(y = t \supset\subset x : Class).$$

Proof. The axioms ($char$), ($N1$)-($N3$), ($V1$), ($V3$), ($V4$) are automatically true by the definition. So only the axioms ($CIG\ def$) and ($CIG\ ind$) remain to be proved. These axioms are valid if the extension of a class notation $c = \mu X_0.\{\vec{a}|A\}$ for a CIG template $\tau = \langle \vec{a}, \vec{v}, \vec{X}, A\rangle$ is the minimal solution of

$$X = \{\vec{x}|[\![A]\!]^{\mathcal{C},\mathcal{E}_{[X/c]}}_{\rho[\vec{x}/\vec{a},c/X_0]}\},$$

where the value of $\mathcal{E}_{[X/c]}$ for c is X and the same as $\mathcal{E}$ for the others. This holds, since by lemma 1 of the previous section, the right hand side of the equation is unchanged even if $\mathcal{E}_{[X/c]}$ is replaced by $\mathcal{E}_{X,\rho}$.

The validity of the additional axiom of decidability of the classes is straightforward, for we may think of the codings involved in the construction as all effective. Note that the axiom is not only effective, but also realizable, for the axiom is of rank 0. Since the class of all classes can be defined by $\{x|app*(cl, x) = t\}$, the restricted system is consistent with the existence of such a class. □

Corollary. *The axiom system of* **PX** *without* ($Join$) *and* ($Product$) *but plus a class constant* CL *and the axiom* $Class(x) \supset\subset x : CL$ *is consistent.*

This axiom system contradicts ($Join$). Let us prove this by an argument due to Feferman 1979. For each class X, we can define its complement by $Cmp(X) =$

$\{x|\neg x : X\}$. So $\mathit{Class}(\mathit{Cmp}(x))$ holds for all $x : CL$. By (Join), we have a class $R_1 = \{\mathit{pair}(x,y)|\neg y : x \ \wedge x : CL\}$. So we can define a class $R = \{x|\neg \mathit{pair}(x,x) : R_1\}$. Then $R : R \supset\subset \neg R : R$. Contradiction.

6.3.2. The full model

Next we construct the interpretation of classes also satisfies (Join) and ($\mathit{Product}$). We denote the functions $\beta(\Pi^*)$ and $\beta(\Sigma^*)$ by $\tilde{\Pi}$ and $\tilde{\Sigma}$, respectively. We suppose two more conditions on the basic function environment:

(iii) The ranges of $\tilde{\Pi}$ and $\tilde{\Sigma}$ are mutually disjoint.

(iv) The ranges of $\tilde{\Pi}$ and $\tilde{\Sigma}$ are disjoint to the image of the set of class constants by *code* and also to the range of $\beta(\mathit{cigcode}(\tau))$ for each CIG template τ.

The construction of classes and extensions of classes, say $\tilde{\mathcal{C}}$ and $\tilde{\mathcal{E}}$, is done by mutual inductive definition as opposite to the construction of $\mathcal{C}$ and $\mathcal{E}$. We do the inductive definition on the following set of partial functions from Obj to $\mathit{Pow}(\mathit{Obj})$:

$$\Delta = \{\langle \mathbf{Cl}, \mathbf{Ext}\rangle | \mathbf{Cl} \in \mathit{Pow}(\mathit{Obj}) \wedge \mathbf{Ext} : \mathbf{Cl} \to \mathit{Pow}(\mathit{Obj})\}.$$

We consider Δ to be an ordered set under the ordering

$$\begin{aligned} &\langle \mathbf{Cl}_1, \mathbf{Ext}_1\rangle \sqsubseteq \langle \mathbf{Cl}_2, \mathbf{Ext}_2\rangle \quad \text{iff} \\ &\quad \mathbf{Cl}_1 \subseteq \mathbf{Cl}_2 \text{ and } \mathbf{Ext}_1 \text{ is the restriction of } \mathbf{Ext}_2 \text{ to } \mathbf{Cl}_1. \end{aligned}$$

Then Δ is a cpo with the bottom $\langle \emptyset, \emptyset\rangle$. We define a monotone function Ψ from Δ to Δ whose any fixed point is a model of **PX** satisfying (Join) and ($\mathit{Product}$). Let $\langle \mathbf{Cl}, \mathbf{Ext}\rangle$ be an element of Δ. Then the value of $\Psi(\langle \mathbf{Cl}, \mathbf{Ext}\rangle)$, say $\langle \mathbf{Cl}_1, \mathbf{Ext}_1\rangle$, is defined as follows:

(I) $x \in \mathbf{Cl}_1$, if x is obtained by one of clauses (1), (2) of the definition of $\mathcal{C}$ by replacing $\mathcal{C}$ by $\mathbf{Cl}$.

(II) $x \in \mathbf{Cl}_1$, if x is of the form $\tilde{\Pi}(c,f)$ or $\tilde{\Sigma}(c,f)$ such that $c \in \mathbf{Cl}$ and $\mathit{App}{*}(1)(f,y) \in \mathbf{Cl}$ holds for all $y \in \mathbf{Ext}(c)$.

(III) Assume x has turned out to be an element of $\mathbf{Cl}_1$ by (I). If x is obtained by the clause (1), then $\mathbf{Ext}_1(x)$ is defined as the clauses (a)-(d) of the definition of $\mathcal{E}$. Otherwise, x is a form $\mathit{cig}_\tau(\rho(v_1), \ldots, \rho(v_n))$ such that $\tau = \langle \vec{a}, \vec{v}, \vec{X}, A\rangle$, and then $\mathbf{Ext}_1(x)$ is the minimal solution of the set equation

$$X = \{\vec{x}| [\![A]\!]^{\mathbf{Cl},\mathbf{Ext}_{X,\rho}}_{\rho[\vec{x}/\vec{a},c/X_0]}\},$$

where $\mathbf{Ext}_{X,\rho}$ is defined as $\mathcal{E}_{X,\rho}$.

(IV) If x is the form obtained by (II), then $\mathbf{Ext}_1(x)$ is defined by

$$\mathbf{Ext}_1(\tilde{\Pi}(c,f)) \overset{\text{def}}{=} \{a|\forall b \in \mathbf{Ext}(c).App{*}(1)(a,b) \in \mathbf{Ext}(App{*}(1)(f,b))\}$$
$$\mathbf{Ext}_1(\tilde{\Sigma}(c,f)) \overset{\text{def}}{=} \{(a\ .\ b)|a \in \mathbf{Ext}(c) \wedge b \in \mathbf{Ext}(App{*}(1)(f,a))\}.$$

It is easy to see that Ψ is a monotone function, since every monotone function on a cpo with a bottom has a minimal fixed point. By the following theorem, we can see that any fixed point of Ψ is a model.

The soundness theorem. *Let $\langle \tilde{\mathcal{C}}, \tilde{\mathcal{E}} \rangle$ be any fixed point of Ψ. Then all of axioms and rules of* **PX** *including* $(Join)$ *and* $(Product)$ *are valid by the semantics* $[\![F]\!]_\rho^{\tilde{\mathcal{C}},\tilde{\mathcal{E}}}$.

The proof, a straightforward consequence of the definition of Ψ, is left to the reader.

The construction of the minimal fixed point of the monotone function Ψ is proved as follows: Define a transfinite sequence $\{X_\alpha\}_\alpha$ such as $X_0 = \bot$, $X_{\alpha+1} = \Psi(X_\alpha)$ and $X_\alpha = \bigsqcup_{\beta<\alpha} X_\beta$ for a limit ordinal β. Then $\{X_\alpha\}_\alpha$ is an increasing sequence. So there is a least ordinal γ smaller than or equal to the cardinal of Δ such that $X_\gamma = X_{\gamma+1}$ and X_γ is the minimal fixed point of Ψ. Feferman used ordinals to construct his model, and the construction of the minimal fixed point of Ψ is essentially the same as his construction of the model. Our construction is a "sugared" Feferman construction.

See Beeson 1985 and Allen 1987 for other constructions of related theories. Allen 1987 gives a constructive proof of existence of a model of Martin-Löf's type theory. His idea is applicable to **PX**. Using $Pow(Obj \times Pow(Obj))$ rather than Δ, we can construct a model of **PX** without modal sign in intuitionistic set theory or type theories.

6.3.3. The model of the impredicative extension

To construct a model of **PX**+CIG2 of 5.2, we need the trick of the "Skolem paradox". (This is essentially nonconstructive.) This is also due to Feferman 1979. In this section, we will construct a model of **PX**+CIG2-$\{(Join), (Product)\}$. The construction of the impredicative full model is obtained by the same method.

First we introduce a modified relative semantics tuned to stratified formulas. It is given by modifying the interpretation of quantifiers over classes so that all class variables range not over the set **Cl**, but the set of *all* subsets of Obj. A bound class variable in a stratified formula appears only in the right hand side of ":", so a bound class variable is never referred to as an object. In other words, the interpretation of a bound class variable in a stratified formula is arbitrary. For example the semantics $\exists X$ is given by

$$\mathbf{Str}[\![\exists X.F]\!]_{\rho}^{\mathbf{Cl},\mathbf{Ext}} \overset{\mathrm{def}}{=} \exists S \subseteq Obj.\mathbf{Str}[\![F]\!]_{\rho[C^*/X]}^{\mathbf{Cl},\mathbf{Ext}_{[S/X]}},$$

where C is an *arbitrary* class constant. Next we associate a Skolem function, say Sk_{ϕ}, with each existential stratified formula, say $\phi(x, X) = \exists Y.F$, so that

$$\text{if } \mathbf{Str}[\![\phi]\!]_{\rho}^{\mathbf{Cl},\mathbf{Ext}}, \text{ then } \mathbf{Str}[\![F]\!]_{\rho}^{\mathbf{Cl},\mathbf{Ext}_{[Sk_{\phi}(\rho(x),\langle \mathbf{Ext}(X),\rho(X)\rangle)/Y]}}.$$

Note that the value of the Skolem function depends on not only the code, i.e., $\rho(X)$, but also the extension of X, i.e., $\mathbf{Ext}(X)$. The existence of such a function is certified by the axiom of choice. Since the number of stratified formulas are countable, we may assume that there are basic function identifiers which code Skolem functions. We assume that they satisfy the following conditions:

(iii) For each stratified formula ϕ, $\beta(Sk_{\phi})$ is a total one-to-one function whose range is disjoint to the image of the set of class constants by *code* and to the range of each $\beta(cigcode(\tau))$.

(iv) If $\phi_1 \not\equiv \phi_2$ then, the ranges of two functions $\beta(Sk_{\phi_1})$ and $\beta(Sk_{\phi_2})$ are disjoint.

We denote $\beta(Sk_{\phi})$ by sk_{ϕ}. The following clause is added to the construction of $\mathcal{C}$:

(3) Let ϕ be a stratified formula with free variables $v_1, \ldots, v_n$. Then for any value assignment ρ with the class condition (i.e., if v_i is a class variable, then $\rho(v_i)$ has been in $\mathcal{C}$), $sk_{\phi}(\rho(v_1), \ldots, \rho(v_n))$ belongs to $\mathcal{C}$.

The clauses of the construction of $\mathcal{E}$ are modified as follows:

(f) If c is $cig_{\tau}(\rho(v_1), \ldots, \rho(v_n))$ and $\tau = \langle \vec{a}, \vec{v}, \vec{X}, A\rangle$, then $\mathcal{E}(c)$ is the minimal solution of the set equation

$$X = \{\vec{x} | \mathbf{Str}[\![A]\!]_{\rho[\vec{x}/\vec{a}, c/X_0]}^{\mathcal{C},\mathcal{E}_{X,\rho}}\}.$$

(g) If c is $sk_{\phi}(\rho(v_1), \ldots, \rho(v_n))$, then $\mathcal{E}(c)$ is $Sk_{\phi}(\rho'(v_1), \ldots, \rho'(v_n))$, where

$$\rho'(v_i) = \begin{cases} \langle \mathcal{E}(\rho(v_i)), \rho(v_i)\rangle & \text{if } v_i \text{ is a class variable,} \\ \rho(v_i) & \text{otherwise.} \end{cases}$$

(h) Otherwise, $Ext(c)$ is an arbitrary subset of Obj.

Then the following lemma holds:

Lemma. *For any stratified formula ϕ and a value environment with the class condition, the modified semantics under $\mathcal{C}$ and $\mathcal{E}$ is equivalent to the original semantics, i.e.,*

$$[\![\phi]\!]_\rho^{\mathcal{C},\mathcal{E}} \quad \textit{iff} \quad \mathbf{Str}[\![\phi]\!]_\rho^{\mathcal{C},\mathcal{E}}.$$

Proof. This is proved by induction on ϕ. Note that any subformula of a stratified formula is again stratified. The point is the change of the interpretation of class quantifiers. For example, let us assume ϕ is $\exists X.F$. The "only if" part is obvious. So let us prove the "if" part. Assume that the right hand side of the theorem holds. Then by the induction hypothesis, the definition of $\mathcal{E}$, and the property of the Skolem function sk_ϕ, we see that

$$[\![F]\!]_{\rho[sk_\phi(\rho(v_1),\ldots,\rho(v_n))/X]}^{\mathcal{C},\mathcal{E}},$$

where $v_1, \ldots, v_n$ are the free variables of ϕ. Since $sk_\phi(\rho(v_1), \ldots, \rho(v_n))$ belongs to $\mathcal{C}$, the left hand side holds. $\Box$

By this lemma, the $\mathbf{Str}[\![A]\!]$ in the clause (f) of the construction of $\mathcal{E}$ may be replaced by $[\![A]\!]$. Hence the restricted soundness theorem holds for the impredicative model. Note that the construction of $\mathcal{C}$ is still effective. So the impredicative extension is still consistent with the axiom of the decidability, and its corollary remains true.

6.4. Satisfication of MGA

We will show the models constructed in this chapter satisfy the minimal graph axiom (MGA) of 2.4.3. Let us reexamine the definition of the first interpreter of 6.1. It is a simultaneous equations with finite many equations and variables. But it may be recognized as simultaneous equation with infinite many variables and equations considering $Def(a)$ is a variable for the function with the function name a. Then the definition equation for Def turns to a collection of infinite many equations. Of course, the minimal solution is unchanged. For simplicity, assume a definition of a function identifier f consists of a single equation $f = \mathit{body}(f)$ and the arity of f is one. Since the semantics of a class defined by a CIG inductive definition is the minimal solution of a monotone operator defined by its CIG template, (MGA) for f states that $Def(f)$ is the minimal solution of the equation of $f = \phi(f)$, where ϕ is the monotone operator on the cpo $[Obj \rightharpoonup Obj]$ defined by $\mathit{body}(f)$. If *app* is included in $\mathit{body}(f)$, then f is *simultaneously* defined with *app*. (MGA) maintains that f is still the minimal solution of a single equation $f = \phi(f)$, even though the graph of *app* in $\mathit{body}(f)$ is interpreted as the graph of *app* which is simultaneously defined with the other functions including f. The following general result suffices to show (MGA) holds.

Lemma 1. *Let $\{D_\lambda\}_{\lambda\in\Lambda}$ be a collection of cpo's with bottoms and let Φ_λ be a continuous function from $\prod_{\lambda\in\Lambda} D_\lambda$ to D_λ. Assume $\langle M_\lambda\rangle_{\lambda\in\lambda}$ is the minimal solution of the simultaneous equations $\{X_\lambda = \Phi_\lambda(\langle X_\lambda\rangle_{\lambda\in\Lambda})\}_{\lambda\in\Lambda}$. Set*

$$\Psi(X) = \Phi_{\lambda_0}(M_X^{\lambda_0}),$$

where $M_X^{\lambda_0}$ is determined by

$$proj_\lambda(M_X^{\lambda_0}) = \begin{cases} X & \text{if } \lambda = \lambda_0 \\ M_\lambda & \text{otherwise} \end{cases}.$$

Then the minimal solution of $X = \Psi(X)$, say N, coincides with M_{λ_0}.

Proof. Since M_{λ_0} is also a solution of $X = \Psi(X)$, $N \sqsubseteq M_{\lambda_0}$ holds. Hence $M_N^{\lambda_0} \sqsubseteq \langle M_\lambda\rangle_{\lambda\in\Lambda}$ holds. So $\Phi_\lambda(M_N^{\lambda_0}) \sqsubseteq M_N^{\lambda_0}$ holds for each $\lambda \in \Lambda$. Hence $\langle M_\lambda\rangle_{\lambda\in\Lambda} \sqsubseteq M_N^{\lambda_0}$ holds. This implies $M_{\lambda_0} \sqsubseteq N$.

7 Implementing PX

In this chapter, we describe an implementation of **PX**. The implementation is based on the formal theory described in earlier chapters, but the actual implementation has extended features. More or less, the extensions are not essential from theoretical viewpoint so that the actual implementation is interpretable in the formal theory of **PX**. The system is implemented by Franz Lisp on a Sun workstation.$^{\text{TM}}$

In 7.1, we will introduce the concept of "hypothesis" to extend the basic logic. It is not an essential extension, but provides a means of backward proof developments. The actual implementation of **PX** is described in 7.2 and 7.3. In 7.4, we will describe a utility by which we can prove hypotheses unproved by **PX**'s proof checker with a more powerful proof checker EKL.

7.1. Hypotheses

As was explained in chapter 4, it is not necessary to fill all details of a proof to extract its execution program. In the course of a proof development, a user often wants to retain many unproved lemmas, as trivial "hypotheses" as in mathematical texts. Giving the full details of a proof in the course of the development, we often lose our main idea of the proof. So we desire a mechanism by which we can retain some "trivial" facts unproved as hypotheses in the course of the development and later can retrieve and certify all hypotheses when we wish to fill in the details of the proof. **PX** has such a feature, called hypothesis. A *hypothesis* is merely a sequent whose conclusion is a rank 0 formula, and any proof may have any finite numbers of hypotheses. To formalize the notion of proofs with hypotheses, it is enough to introduce the following two inference rules:

$$(hypI) \qquad \Gamma \Rightarrow F \quad (F \text{ is of rank } 0)$$

$$(hypE) \qquad \frac{\Gamma_1 \Rightarrow F_1 \quad \Gamma_2 \Rightarrow F_2}{\Gamma_1 \Rightarrow F_1}.$$

The rule (*hypI*) means we may use any hypothesis as a new axiom. To explain (*hypE*), we define the *hypotheses* of a proof.

Definition 1. Let P be a proof ends with a rule r. Then the hypotheses of P, say $hyp(P)$, are defined by

(1) If r is (*hypI*), then $hyp(P)$ is $\{\Gamma \Rightarrow F\}$.

(2) If r is (*hypE*), let P_1 and P_2 be the left and right immediate subproof of P. Then we set

$$hyp(P) = (hyp(P_1) - \{\Gamma_2 \Rightarrow F_2\}) \cup hyp(P_2).$$

(3) If r is neither (*hypI*) nor (*hypE*), then $hyp(P)$ is $hyp(P_1) \cup \ldots \cup hyp(P_n)$, where $P_1, \ldots, P_n$ are the immediate subproofs of P.

If a proof P has hypotheses, i.e., $hyp(P) \neq \emptyset$, then P is called a *hypothetical proof* or an *incomplete proof*. Otherwise, it is called a *complete proof*. A sequent is a theorem iff a complete proof ends with it.

A proof of the form $S_1/S_2/S_3$ may be thought to be constructed in two ways $(S_1/S_2)/S_3$ and $S_1/(S_2/S_3)$. The former is a proof by forward reasoning and the latter is by backward reasoning. But the official inductive definition of proof trees does not allow an "incomplete" proof like (S_2/S_3). In this sense, backward reasoning is forbidden in traditional formal systems. Regarding S_2 as a hypothesis, i.e., a proof of S_2 created by (*hypI*), we think (S_2/S_3) is a hypothetical proof and we get a complete proof $S_1/(S_2/S_3)$ by applying (*hypE*). That is to say, (*hypI*) reserves a point (leaf) of a proof tree and another proof tree is grafted on the tree at the point by (*hypE*). Backward reasoning is a way to build a proof tree from root to leaves; in this sense (*hypI*) and (*hypE*) provide a means of backward reasoning. Note that to do "one step backward inference" we have to make a proof from new goals created by (*hypI*) by the corresponding forward inference rule and then apply (*hypE*). This takes many steps, so if a proof is built mainly by backward reasoning our mechanism is not useful. But actual proof writing in mathematics mainly proceeds in the forward direction. Although one step backward reasoning may be useful as a tool for proof *development*, as in Edinburgh LCF (Gordon, Milner, and Wadsworth 1979, Constable et al. 1986), it seems not too useful for proof *writing*.

By attaching the realizer of the left upper sequent to the lower sequent, we can realize the rule (*hypE*). The following obviously holds:

Proposition 1. *Let P be a hypothetical proof of $\Gamma \Rightarrow F$ and let r be a program extracted from P. Then r is a realizer of $\Gamma \Rightarrow F$, if all of the hypotheses of P are valid.*

Furthermore, the realizer extracted from a proof P_1 that is obtained by eliminating hypotheses of a proof P by successive applications of (*hypE*) is identical to the realizer extracted from P. So proof-program development in **PX** may proceed in one of the following ways, and the results are completely the same.

Approach A.

0) Give a specification as a sequent.
1) Make an incomplete proof of the specification.
2) Extract a realizer of the specification from the proof.
3) Prove the hypotheses of the proof.

Approach B.

0) Give a specification as a sequent.
1) Make an incomplete proof of the specification.
2) Complete the proof eliminating hypotheses by (*hypE*).
3) Extract a realizer of the specification from the completed proof.

A termination condition can be stated as a hypothesis, as we saw in chapter 4; this is the hardest part of verification. So we normally put it in the hypotheses and take approach A, for we can see the program before we finish the cumbersome verification of the termination condition. Since hypotheses are rank 0 sequents, it is possible to prove them by means of another proof checker, insofar as the checker is faithful to the semantics of **PX**. Namely, step 3 of approach A may be done by another proof checker. To do so we have to make a translator of languages and a theory in **PX**, which the proof is developed in, to a servant proof checker. (In proof development, a theory is built up on plain **PX** by declaring functions and so on. See the next section.) Having such conversion method, approach A is modified as follows.

Approach C

0) Give a specification as a sequent.
1) Make an incomplete proof of the specification.
2) Extract a realizer of the specification from the proof.
3) Translate the theory and hypotheses into the servant proof checker.
4) Prove the translated hypotheses in the theory by using the servant proof checker.

We have implemented a translator that translates a theory and hypotheses of **PX** into a powerful proof checker EKL; this will be described in 7.4. An actual program-proof development by approach C with EKL translator will be given in appendix B.

7.2. Proof checker

In this section we briefly describe the proof checker of **PX**. We will not give accounts of all functions supported by **PX**.

7.2.1. Formulas

First we give the abstract syntax of formulas. Table 1 shows the correspondence of the abstract syntax and the concrete syntax. For readability, any formula may

Table 1

Concrete syntax	Abstract syntax
app, $app*$	`apply`, `funcall`
$pair$, fst, snd, $list$	`cons`, `car`, `cdr`, `list`
$atom$, $equal$	`atom`, `equal`
suc, prd	`add1`, `sub1`
0 ,t ,nil, V, N, Atm, T, $quote(\alpha)$	`0` ,`t` ,`nil`, `V`, `N`, `Atm`, `T`, (`quote` α)
$f(e_1,\ldots,e_n)$	$(f\ e_1\ \ldots\ e_n)$
$cond(e_1,d_1;\ldots,e_n;d_n)$	(`cond` $(e_1\ d_1)\ldots(e_n\ d_n)$)
$\lambda(x_1,\ldots,x_n)(e)$	(`lambda` $(x_1\ \ldots\ x_n)\ e$)
$\Lambda(x_1=e_1,\ldots,x_n=e_n)(f)$	(`Lambda` $((x_1\ e_1)\ldots(x_n\ e_n))\ f$)
$let\ p_1=e_1,\ldots,p_n=e_n\ in\ e$	(`let!` $((p_1\ e_1)\ldots(p_n\ e_n))\ e$)
$\top$, $\bot$	`TRUE, FALSE`
$[e_1,\ldots,e_n]:e$	$e_1\ \ldots\ e_n$ `:` e
$e_1=e_2$	e_1`=`e_2
$E(e)$	`E` e
$Class(e)$	`CL` e
$F_1\wedge\ldots\wedge F_n$	F_1`&`$\ldots$`&`F_n
$F_1\vee\ldots\vee F_n$	F_1`+`$\ldots$`+`F_n
$F_1\supset F_2$	F_1`->`F_2
$F_1\supset\subset F_2$	F_1`<->`F_2
$\neg F$	`-`F
$\Diamond F$	`$`F
$e_1\to F_1;\ldots;e_n\to F_n$	(`COND` $(e_1\ F_1)\ldots(e_n\ F_n)$)
$Sor(e_1,\ldots,e_n)$	(`OR` $e_1\ \ldots\ e_n$)
$Case(e,F_1,\ldots,F_n)$	(`CASE` $e\ (F_1)\ \ldots\ (F_n)$)
$\nabla p_1=e_1,\ldots,p_n=e_n.F$	(`LET` $(p_1\ e_1)\ldots(p_n\ e_n))(F)$
$\forall[x_1,\ldots,x_l]:e_1,\ldots,[y_1,\ldots,y_m]:e_n.F$	(`UN` $(x_1\ \ldots\ x_l):e_1\ldots(y_1\ \ldots\ y_m):e_n)(F)$
$\exists[x_1,\ldots,x_l]:e_1,\ldots,[y_1,\ldots,y_m]:e_n.F$	(`EX` $(x_1\ \ldots\ x_l):e_1\ldots(y_1\ \ldots\ y_m):e_n)(F)$

be surrounded by parentheses. But the print routine of **PX** always deletes a repeated grouping of parentheses, so that even if one types in ((`COND`...)), **PX** prints (`COND`...).

When the length of a tuple variable is just one, it is treated as a single variable. And the restriction by V may be omitted. So one may type in (`UN x`)(F) instead of (`UN (x) : V`)(F). Furthermore, (`UN x : C y : C`)(F) may be abbreviated as (`UN x y : C`)(F). `CASE` and `OR` are treated as "abbreviations", i.e., the read routine translates these into conditional formulas, and the print routine abbreviates conditional formulas into these as much as it can. **PX** allows to use defined predicates and predicate variables. These are explained in 7.2.3.

The abstract syntax is comprised of external forms of formulas. When these are inputs, they are translated to internal forms. Every internal form is just an S-expression, e.g., $x = y$ is represented by (%= x y).

To let **PX** recognize an input sequence as a formula, one has to surround it by braces, as {x=y}. The left brace is a character macro which reads the input sequences until the next right brace and returns the internal form of the formula. So the internal form of $x = y$ is inputed by typing in '{x=y}.

Variables and constants in the abstract syntax are as follows:

$$\langle\text{variable}\rangle := \langle\text{total variable}\rangle | \langle\text{partial variable}\rangle$$

$$\langle\text{total variable}\rangle := \langle\text{individual variable}\rangle | \langle\text{class variable}\rangle$$

$$\begin{aligned}\langle\text{individual variable}\rangle := {} & \langle\texttt{a-z}\rangle \text{ except } \texttt{t} | \langle\texttt{a-z}\rangle\langle\text{numeral}\rangle \\ & |\$\langle\text{individual symbol}\rangle\langle\text{symbol}\rangle \\ & |\langle\text{individual variable}\rangle *\end{aligned}$$

$$\begin{aligned}\langle\text{class variable}\rangle := {} & \langle\texttt{A-Z}\rangle \text{ except } \langle\texttt{N}|\texttt{V}\rangle|\$\langle\texttt{A-Z}\rangle\langle\text{symbol}\rangle|\langle\texttt{A-Z}\rangle\langle\text{digit string}\rangle \\ & |\langle\text{class variables}\rangle *\end{aligned}$$

$$\langle\text{partial variable}\rangle := ?\langle\text{symbol}\rangle$$

$$\langle\text{identifier}\rangle := \langle\text{individual identifier}\rangle | \langle\text{class identifier}\rangle$$

$$\begin{aligned}\langle\text{individual identifier}\rangle := {} & \langle\texttt{a-z}\rangle\langle\texttt{a-z}|\texttt{A-Z}\rangle\langle\text{symbol}\rangle|\texttt{t}|\langle\text{numeral}\rangle \\ & |(\texttt{quote } \langle\text{S-expression}\rangle)\end{aligned}$$

$$\langle\text{class identifier}\rangle := \langle\texttt{A-Z}\rangle\langle\texttt{a-z}|\texttt{A-Z}\rangle\langle\text{symbol}\rangle|\texttt{N}|\texttt{V}|\texttt{T}$$

$$\begin{aligned}\langle\text{symbol}\rangle := {} & \texttt{0}|\texttt{1}|\ldots|\texttt{9}|\texttt{A}|\texttt{B}|\ldots|\texttt{y}|\texttt{z} \\ & |\langle\text{symbol}\rangle\langle\text{symbol}\rangle \\ & |\langle\text{symbol}\rangle - \langle\text{symbol}\rangle \\ & |\langle\text{symbol}\rangle\_\langle\text{symbol}\rangle\end{aligned}$$

$$\langle\text{individual symbol}\rangle := \langle\text{symbol}\rangle \text{ except } \langle\texttt{A-Z}\rangle$$

$$\langle\text{digit string}\rangle := \texttt{0}|\texttt{1}|\ldots|\texttt{9}|\langle\text{digit string}\rangle\langle\text{digit string}\rangle$$

$$\langle\text{numeral}\rangle := \texttt{0}|\langle\texttt{1}|\ldots|\texttt{9}\rangle\langle\text{digit string}\rangle$$

Except for these, **PX** creates some individual variables and individual identifiers during extraction. Although they are noted as variables and identifiers by **PX**, users are not allowed to use them unless **PX** has created them.

These syntactic categories are not the syntactic categories of the DEF-system. An individual identifier may be either an individual constant or a function identifier. After one declares which it is, or after one places it in the proper position in the input formula, it belongs to one of theses exclusively. This will be explained in 7.2.6. To know what an individual belongs to, a command `syncat`, which returns the syntax category of each atom, is available. Similarly, a class identifier may be either a class constant or a function identifier. The arity of a defined function is determined by its declaration. Some function identifiers and constants are predeclared, e.g., `car` is a function identifier, `t` is an individual constant, and `N` is a class constant. Note that it is not necessary to declare variables. When renaming of bound variables is necessary for the substitution of expressions, **PX** adds enough numbers of "`*`" to the end of variables.

7.2.2. Proofs

A *proof* is also an S-expression. A proof is a list beginning with the atom `%%proof%%`. A list beginning with `%%proof%%` is called a *pseudo proof.* Insofar as **PX** is used legally, any pseudo proof has the structure of a proof. The second and third items of a proof are the conclusion and the list of assumptions. In **PX** mode, the default mode of **PX**, any illegal input is detected by the top level and is not executed. So pseudo proofs are always proofs. But in Lisp mode, one may execute any lisp commands, so it is child's play to make an S-expression which is a pseudo proof but not a proof. For example, the value of `(list '%%proof%% '%FALSE nil)` is a pseudo proof of `FALSE` without any assumptions. **PX** does not support any method to check if an S-expression is an actual proof. So the only way to ensure that one has not built pseudo proofs which are not proofs is to use **PX** only in the **PX** mode. By the form of the prompt, one may ascertain that one has fallen into the Lisp mode.

7.2.3. Top level and commands

The top level of **PX** is a modification of the CMU top level of Franz Lisp. So one may omit the outermost parentheses at the top level; the top level retains the events, which may be redone (see Foderaro, Sklower, and Layer 1984). The read and print routines of **PX** are almost the same as those of the CMU top level of Franz Lisp, and the brace macro "{" for reading external formulas is supported, as mentioned above, and read-eval routines support a checking mechanism for legal commands.

A command is any program of Lisp. **PX** checks if it may create an illegal pseudo proof and then executes it as a Lisp program only when it is safe. Such a safe program is called a *legal command.* A legal command is a command whose execution does not violate the following condition: every object created in **PX** represents a proof insofar as it is a `cons` cell and its `car` part is `%%proof%%`. For example a destructor, e.g., `car`, is safe, for it does not create new cells. On the contrary, `cons` is dangerous. So a command using `cons` is illegal. But list constructors inevitably use Lisp as metalanguage in the sense of ML of Edinburgh LCF. So **PX** supports the *safe constructors* `sfcons`, `sflist` etc. These constructors never create a cell whose `car` part is `%%proof%%`. (The `sfcons` stands for *safe cons*.) Other functions besides constructors that may violate the condition are the loading function and functions assigning a value or a function definition to atoms. For the former, a legal function `sfload` is available. It checks that each input from a file is legal. Its actual use is shown in 7.2.6. The other functions will also be discussed in 7.2.6.

The print routine supports pretty printing of formulas and proofs. If an S-expression is a formula, the print routine prints it as an external formula surrounded by braces. If an S-expression is a proof, the print routine prints a sequence of representing dots and the yield sign "`]-`" followed by the conclusion, as Edinburgh LCF does. A printed proof is also surrounded by braces, and if it is a hypothetical proof, then an asterisk follows the left brace. The following shows how this works:

```
1:assume {(add1 x) = (sub1 x)}
{.]- (add1 x) = (sub1 x)}          ; A proof with an assumption.
2:(deprf prf1 it)                  ; Assign it to the atom  prf1.
{.]- (add1 x) = (sub1 x)}
3:display-thm prf1                        ; Display the sequent of the proof.
prf1.
  {(add1 x) = (sub1 x)}
    from
  [1] {(add1 x) = (sub1 x)}

t
4:(deprf prf2 (hypI '{x = (cons x x)}  ; Introduce a hypothesis.
                    '{(car x) = x}
                    '{(cdr x) = x}))
{*..]- x = (cons x x)}                  ; An asterisk is observed.
5:display-thm it        ; The value of the last event was assigned to the atom
it.
it.
  {x = (cons x x)}
    from
  [1] {(car x) = x}
  [2] {(cdr x) = x}
  with 1 hypotheses           ;  display-thm tells there is a hypothesis.

t
6:(deprf prf3 (conjI prf1 prf2))  ; Conjunction introduction.
{*...]- (add1 x) = (sub1 x) & x = (cons x x)}
7:(display-thm prf3)
prf3.
  {(add1 x) = (sub1 x) & x = (cons x x)}
    from
  [1] {(add1 x) = (sub1 x)}
  [2] {(car x) = x}
  [3] {(cdr x) = x}

  with 1 hypotheses

t
9:(showhyp prf3)                ; Display the hypotheses of  prf3.

 Hypothesis
  {x = (cons x x)}
    from
  [1] {(car x) = x}
  [2] {(cdr x) = x}

t
```

In **PX** mode, the prompt is an event number followed by ":". So the first thing one sees when **PX** is loaded is "1:" as above. In Lisp mode, the prompt is an event number followed by "." as in the CMU top level. After you enter in Lisp mode once, the prompt of **PX** mode turns to an event number followed by ";". The function `mode` changes the mode.

The following shows the changes of mode:

```
1:mode                            ; Mode is PX.
PX
2:assume {FALSE}                  ; Create a proof of {⊥} ⇒ ⊥.
{.]- FALSE}
3:display-thm it                      ; Display the sequent of  it.
it.
  {FALSE}
    from
  [1] {FALSE}

t
4:vl 2                              ; Get the value of the event 2.
{.]- FALSE}
5:pp it                             ; Print it to see the internal form.
(setq it '(%%proof%% (%FALSE) ((%FALSE)) (%assume (0 (%FALSE)))))
t
6:'(%%proof%% (%FALSE) ())          ; Make an illegal proof.
'(%%proof%% (%FALSE) nil)
Illegal input for PX                ; The attempt failed.
nil
7:mode lisp                         ; Change to Lisp mode.
LISP
8.'(%%proof%% (%FALSE) ())          ; Make an illegal proof.
{]- FALSE}                          ; It succeeded, this time.
9.display-thm it                    ; Display it.
it.
  {FALSE}

t
10.(setq fff (vl 8))                ; Keep the illegal proof.
{]- FALSE}
11.mode px                          ; Again PX mode.
PX
12;'(%%proof%% (%FALSE) ())         ; So you can't remake the illegal one.
'(%%proof%% (%FALSE) nil)
Illegal input for PX
nil
13;fff                              ; But the old one is still kept.
{]- FALSE}
```

Observe that after entering Lisp mode once, one may have a bad proof, even if one has again returned to **PX** mode.

The function `con` returns the conclusion of a proof, the function `asp` returns a list of the assumptions of a proof and the function `get_hyp` returns a list of the hypotheses of a proof. A hypothesis $\{A_1, \ldots, A_n\} \Rightarrow F$ is represented as a list $(F\ (A_1\ \ldots\ A_n))$. To display hypotheses, a function `showhyp` is available as shown above.

7.2.4. Inference rules

Since a proof is an S-expression, any inference rule that is a Lisp function returns a proof. For example, (*assume*) is implemented as a function `assume` such that

$$(\texttt{assume}\ F) = \{F\} \Rightarrow F \quad (F \text{ is a formula}),$$

and the conjunction introduction ($\wedge I$) is a function `conjI` such that

$$(\texttt{conjI}\ P_1\ \ldots\ P_n) = \frac{P_1, \ldots, P_n}{F_1 \wedge \ldots \wedge F_n}\ (\wedge I).$$

So the value of `(conjI (assume '{x=`y`}) (assume '{y=z}))` is a proof of the sequent

$$\{\mathtt{x} = \mathtt{y},\ \mathtt{y} = \mathtt{z}\} \Rightarrow \mathtt{x} = \mathtt{y}\,\&\,\mathtt{y} = \mathtt{z}.$$

Almost all inference rules are implemented as functions in the manner described above, respecting their concrete description. But there are some exceptions. The axioms about the modal operator are not implemented separately, but as part of a tautology checker. **PX** has a which returns a proof of a *rank 0 formula* if the following procedures succeed: (i) regarding the modal operator as double negation, eliminate all successive pairs of negations, (ii) expanding conditional formulas to the disjunctive equivalent in 2.3.4, and regarding any quantified formula as a propositional variable and two quantified formulas as the same propositional formulas when they are α-convertible, check if it is a tautology. The tautology checker respect only propositional logic, although it call `axiom?` below to check if formulas are axioms. The following shows a use of the tautology checker `TAUT`.

```
1:deGP ^Prd 1 0  ; Declare ^Prd as a rank 0 predicate of arity 1. See 7.2.6.
^Prd
2:TAUT {$ ^Prd(x) <-> ^Prd(x)}    ; Axiom (◇3).
{]- $ ^Prd (x) <-> ^Prd (x)}
3:TAUT {$ ^Prd(x) <-> - - ^Prd(x)}    ; Axiom (◇1).
{]- $ ^Prd (x) <-> - - ^Prd (x)}
4:TAUT {$(UN x)(^Prd(x)) <-> (UN x) ($ ^Prd(x))}  ; Axiom (◇2).
{]- $ (UN x) (^Prd (x)) <-> (UN x) ($ ^Prd (x))}
5:TAUT {- (EX x)(- ^Prd(x)) <-> (UN x)(^Prd(x))}
nil          ;  TAUT doesn't know predicate calculus.
```

Another exception concerns is a data base of axiom schemas. One can ask **PX** if a formula is its axiom schema by a query function `axiom?` as follows:

```
1:axiom?  {E x}   ;  axiom? knows a total variables has a value.
{]- E x}
```

```
2:axiom?  {E ?1}  ;  axiom? doesn't know about this.
Error in axiom?:  I don't know {E ?1}
3:axiom?  {(cons ?1 ?2) :  Dp} !(assume '{E ?1}) !(assume '{E ?2})
{..]- (cons ?1 ?2) :  Dp}
4:display-thm it
it.
  {(cons ?1 ?2) :  Dp}  ;  (cons ?1 ?2) is a dotted pair,
    from                 ; under the following assumptions.
  [1] {E ?1}
  [2] {E ?2}
t
5:axiom?  {(car (cons x y)) = x}
{]- (car (cons x y)) = x}
6:axiom?  {E ?1} !(assume '{E (cons ?1 0)})
{.]- E ?1}
7:display-thm it
it.
  {E ?1}
    from
  [1] {E (cons ?1 0)}
t
8:(deprf th (axiom?  '{?1 = ?1}))
{]- ?1 = ?1}
9:(exI '{(EX x)(x = x)} th (assume '{E (cons ?1 x)}))
{.]- (EX x) (x = x)}   ;  axiom? is called in (∃I).
10:display-thm it
it.
  {(EX x) (x = x)}
    from
  [1] {E (cons ?1 x)}
t
```

7.2.5. Inference macros

PX provides macros by which you can write natural programs to build up proofs. These programs are then expanded to programs written in terms of the inference rules described above and executed. So we call them *inference macros*. The example of appendix B is written in these inference macros. In this subsection, we will give an overview of them.

Let e be a program whose value is a list of proofs. Then (`We_see` F e) is expanded to an appropriate inference rule. For example, if the values of e_1 and e_2

are proofs whose conclusions are A and B, then `(We_see` $\{A$ `&` $B\}$ `(sflist` e_1 e_2`))` is expanded to `(conjI` e_1 e_2`)`. When you wish to specify what rule is used, you can say, e.g., `(We_see` F `by conjI` e`)`. The function `sflist` does not look natural. **PX** provides some *connectives* like `since`, `for`, `by`, `in_the_following`, etc., which are aliases of `sflist`. So the above example may be restated as `(We_see` $\{A$ `&` $B\}$ `(for` e_1 e_2`))`. The connectives that start with an upper case letter: `By`, `Since`, `In_the_following`, etc. provide a way to change the order of sentences in proofs. For example, `(By` e `(we_see` F`))` is expanded to `(we_see` F `(by` e`))`. (`we_see` is an alias of `We_see`. All the macros except some connectives have such aliases.)

The macro `proofs` is similar to prog of Lisp. (`@` is a synonym for `proofs`.) `(proofs` $e1$ ... e_n`)` equals `(prog` $e1$ ... e_n`)`, but in the program e_{i+1} the value of the previous program e_i can be referred as `the_previous_fact`. On the other hand, `the_previous_facts` stands for the list of the values of the previous expressions.

`Let` and `Set` are aliases of `sfsetq` explained below. When `(We_prove (A)` F e`)` is evaluated, the formula F is bound to the atom `A` then e is evaluated. So one can refer to F by the label `A`. Furthermore, it applys the rule (*alpha*) to F and the value of e. Thus, if the value of e is a proof whose conclusion is α-convertible to F, then the value of the program is a proof of F. The label `(A)` may be omitted. Then only the adjustment by (*alpha*) is done.

In the mathematical text, the elimination rules like existential elimination introduce *local assumptions*. When we prove $\exists x.A \Rightarrow C$ by existential elimination, we introduce a local assumption A and prove C under the assumption. But the official rule ($\exists E$) does not allow such an inference. We must built proofs of $\{\exists x.A\} \Rightarrow \exists x.A$ and $\{A\} \Rightarrow C$, then apply the rule. This is not quite a natural way of proving theorems. The above proof can be written as follows by a macro `We_assume`:

```
(We_assume there_exists x such_that (Asp1) F e).
```

When this expression is evaluated, a proof of $\{F\} \Rightarrow F$ is bound to `Asp1`, then e is evaluated. So one can refer to the local assumption F by `Asp1` in e. But this does not exhaust the features of the macro. **PX** has an *assumption stack* in which local assumptions are kept. The macro also pushes the local assumption on the stack. When one says `(Obviously` B G_1 ... G_m`)` in e, it is expanded to `(hypI` B F_1 $\ldots F_n$ G_1 ... G_m`)`, where $F_1, \ldots, F_n$ is the content of the assumption stack. After evaluating e, the local assumption pushed by the macro pops up. See examples in appendix B.

When the major premise of ($\exists E$) is not an assumption but a proved fact, then one may say

$$\texttt{(We\_may\_assume there\_exists x such\_that (A)}\ F\ e_1\ e_2\texttt{)}$$

Then the value of e_2 is used as the major premise of $(\exists E)$. By combining connectives such as `By`, etc., with this, one can say `(By` e_2 `(we_may_assume ...` e_1`))`. The macros for the elimination rule for ∇ are also `We_assume` and `We_may_assume`. The macros understand the rule should be $(\exists E)$, if the keyword `there_exists` exists. If it is absent and a declaration `(Asp1)` p `matches` e exists, then $(\nabla\exists E)$ is used.

The macros for $(\vee E)$ and $(\rightarrow \vee E)$ are `By_cases`. The keywords by which the macro decides which rule should be used are `or` for $(\vee E)$ and `otherwise` for $(\rightarrow \vee E)$. See appendix B.

To push assumptions onto the assumption stack, one may say `(Suppose (A)` F e`)`. Then F is pushed onto the stack, the assumption is bound to the label `A`, and e is evaluated. `(We_assume (A)` F e`)` is equivalent to the above. Namely, `We_assume` turns to be an alias of `Suppose`, if the keywords mentioned above are absent.

Since the inference macros look like English phrases, it is child's play to print a program of **PX** written by inference macros in TEX. **PX**'s TEX *translator* deletes underscores of macro names and translates formulas in abstract syntax to TEX commands representing formulas in concrete syntax. Then the resulted TEX file can be formated by TEX. An example will be found in appendix B. Note that inference macros are expanded to basic inference functions, so **PX** cannot translate internal forms of proofs to TEX commands. It is possible to defined new inference macros in **PX**. Then the TEX translator print them by using a default format. So to print them beautifully, a user has to specify how to print them by means of TEX commands.

7.2.6. PX evaluator and declarations

PX can be considered to be based on a regular DEF-system. A regular DEF-system consists of an infinite number of function definitions. But a DEF-system created by the actual **PX** always has only a *finite* number of function definitions. We may regard any environment created with the proof checker of **PX** to be a finite subsystem of a regular DEF-system. The environment created by **PX** is called a *theory* and it consists of a finite number of declarations. A theory is not an environment in Lisp on which **PX** is implemented. So even if one declares a function in a theory, it is not available as a Lisp function. Since a theory is different from the environment of Lisp, users are allowed to declare a function with a function name which **PX** uses. For example, in the theory of the experiment in appendix B, we declare a function `con` which returns the conclusion of a proof of the propositional logic defined in the theory. This does not change the definition of the function `con` of **PX**, which returns the conclusion of a proof of **PX**. The **PX** evaluator, `px-eval`, evaluates expressions of **PX** under the environment of theory. `px-eval` may have less than two arguments. `(px-eval` e a`)` evaluates the expression e under the assignment a, which assigns values to free variables of e. An assignment is a list of lists of the form (variable value). `px-eval` has a *current environment* and an expression e is evaluated under the current environment by `(px-eval` e`)`. One can assign a value of the expression e to a variable x of the current environment by `(px-eval-setq` x e`)`. Note that this does not change the theory. Even if one assigns a value to a variable of the current environment of `px-eval`, one cannot prove that the variable is equal to the value. The assignment of values to variables in the current environment is allowed only for the convenience of running extracted functions. By `(px-eval-import` x y`)`, one can import the value of the variable y of the proof checker to the current environment. Invoking the command `(px-eval)`, one enters into the interactive mode of the **PX** evaluator. Then each command is evaluated under the current environment. In the interactive mode of **PX** evaluator, one changes the current environment by `setq`, which is available only on the top level of the interactive mode. One can exit from the **PX** evaluator top level by `(exit)`, and then the current environment is retained.

Declarations are classified to *constant declarations*, *function declarations*, or *predicate declarations*. Constant declaration declares identifiers as constants in the sense of DEF system. Function declaration declares identifiers as function identifiers in the sense of DEF system. Function and constant declarations are also classified as *class declarations* and *individual declarations*. Predicate declarations are further classified as *generic predicate declarations*, and *user defined predicate declarations*. So there are six kinds of declarations, i.e., individual constant declaration, individual function declaration, class constant declaration, class function declaration, generic predicated declarations, and user-defined predicate declarations. Declarations are exclusive, i.e., an identifier once declared cannot be redeclared. This restriction retains the consistency of proofs and an environment built by declarations. A declaration is done via one of the following functions: `deCONST, deFUN, deEXFUN, deCIG, deECA, deDP, deGP` or the read routine of **PX**. The declarations done by the read routine are called *incomplete declarations*. Even if the individual identifier `ff` is not declared, one can use it in one's formula. If one inputs '{`(ff x) = y`}, then the read routine declares `ff` as a function identifier with arity 1. If one inputs '{`ff = y`} instead, then the read routine declares `ff` as a constant. Such declarations are called *incomplete declarations*, for they does not determine the definition of identifiers. When an identifier has an incomplete declaration, the definition of the identifier may be declared by `deFUN`, `deCIG`, deECA, or deCONST according to its incomplete declaration. Note that deEXFUN cannot be used. At extraction, the definition of an incomplete identifier is an appropriate default value.

Numerals and the identifier of the form (quote α) are constants without declarations. And some identifiers t, nil, etc., are predeclared.

The function deCOSNT declares an individual identifier as a constant. So declarations done via deCONST are always individual constant declarations. Invoking (deCONST xx v), the individual identifier xx is declared as an individual constant with the *value* v. If xx is not an individual identifier, the declaration is denied. After the declaration, the axiom xx = 'v is added to the theory. Invoking (axiom 'xx) one obtains a proof of this axiom. If the body of deCONST is empty, e.g., (deCONST xx), then xx is declared to be a constant and the axiom E xx is added to the theory. The declaration (deCOST (xx 1) (yy) (zz t)) declares xx, yy, and yy at once.

The function deFUN declares a function. For example, append is declared as

```
(deFUN append (x y)
       (cond (x (cons (car x) (append (cdr x) y))) (t y))).
```

A function declaration via deFUN is always an individual function declaration. A function declaration must be lexical, i.e., all free variables of the function body must appear in the argument list. Incremental programming is *not* available in **PX**, i.e., all functions in the body of deFUN must have been declared. Simultaneous function declaration is available, e.g., even and odd can be declared as follows:

```
(deFUN (even (x) (cond ((equal  x  0)  t) ( t ( odd ( sub1  x)))))
       (odd (x) (cond ((equal x 1) t) (t (even (sub1 x)))))).
```

The axioms added by a function declaration are definition equations of the declared functions. For example, after the above declaration, one obtains a proof of

```
(odd x)=(cond ((equal x 1) t) (t (even (sub1 x))))
```

by invoking (axiom 'odd). (deFUN foo (x y)) will do an incomplete declaration.

The derived rule of (*choice*2) of 2.5 is implemented as an individual function declaration. When P is a proof whose sequent satisfies the condition of (*choice*2), then its choice function is declared by

(deEXFUN foo (x1 ... xn) 'P).

When this is invoked, a program extracted from P and a function declaration of `foo` is done by `deFUN`. So `(axiom 'foo)` returns a proof whose conclusion is the definition of `foo`. When **PX** creates some new functions for CIG induction in P, then they are also declared by `deFUN`. `(choice-rule 'foo)` returns a proof of the lower sequent of the rule (*choice*2).

A CIG definition is done by `deCIG` or `deECA`. `(deECA` x y_1 ... y_n`)` is just a macro for `(deCIG` x `(t` y_1 ... y_n`))`. The abstract syntax of ($CIG\ dec_1$) of 3.2 is as follows:

$$
\begin{array}{l}
\texttt{(deCIG } \{\texttt{xx : C}\} \texttt{ (in D)} \\
\qquad\qquad (e_1\ \phi_{1,1}\ \ldots\ \phi_{1,q_1}) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad (e_n\ \phi_{n,1}\ \ldots\ \phi_{n,q_n})\texttt{)}.
\end{array}
$$

The class notation `C` may be a class identifier or an expression of the form (f $\vec{x}$), where f is a class identifier and $\vec{x}$ is a sequence of total variables. In the former case, the declaration is a class constant declaration, which declares the class identifier as a class constant, and in the latter it is a class function declaration, which declares the class identifier as a function. The axiom maintaining `C` is the minimal fixed point is available by `(axiom 'C)`. The axiom maintains `C` is a class is available by the function `axiom?`. When one declares `C` by CIG, `(axiom? '{CL C})` will return a proof of `CL C`. The following are some actual uses of these functions:

```
1.(deCIG {x : (Tree X)}     ; Declare the class of binary trees.
          ((atom x) {x : (Tree X)})
          (t {(car x) : (Tree X)} {(cdr x) : (Tree X)}))
(Tree X)
2.axiom Tree                ; The axiom (CIG def) for (Tree X).
{]- (COND ((atom x) x : (Tree X))
          (t (car x) : (Tree X) & (cdr x) : (Tree X)))
        <-> x : (Tree X)}
3.axiom? {CL (Tree X)}           ; (Tree X) is a class.
{]- CL (Tree X)}
4.(deCIG                    ; Simultaneous definition of Even and Odd.
     ({x : Even} (in N) ((equal x 0) {TRUE})
                        (t {(sub1 x) : Odd}))
     ({x : Odd}  (in N) ((equal x 0) {TRUE})
                        (t {(sub1 x) : Even})))
(Even Odd)
9.axiom Even                ; The axiom (CIG def) for Even.
{]- x : N &
       (COND ((equal x 0) TRUE) (t (sub1 x) : Odd)) <-> x : Even}
```

```
10.axiom Odd                    ; The axiom (CIG def) for Odd.
{]- x :  N &
       (COND ((equal x 0) TRUE) (t (sub1 x) :  Even)) <-> x :  Odd}
```

The function implementing the inference of (*CIG ind*) is `cigIND`. The following is a part of an actual session which created a hypothetical proof of (B4) of 4.3.2.1. It also illustrates a use of `sfload`.

Programs create hypothetical proofs of (B1) and (B2) of 4.3.2.1 and assign them to `BASIS` and `STEP` are in a file named `divitr.px`, whose first lines are as follows:

```
; These are proofs of (B1) and (B2) of 4.3.2.1.

(deCIG {a :  (DD b)} (in N)
     ((lessp a b) {TRUE}) (t {(diff a b) :  (DD b)}))

(sfsetq Goal
    '{(EX q r :  N) (a = (plus (times b q) r) & (lessp r b) :  T)})

(deprf BASIS
  (exI Goal
        (deprf Lemma1
                                    ⋮
```

So first we `sfload` them and create a proof of (B4) by the CIG induction rule `cigIND`.

```
1:sfload divitr                          ; Load the file of the proof.
[loading divitr.px in PX mode]
t
2:display-thm BASIS STEP                 ; Display basis and induction step.
BASIS.
  {(EX q r :  N) (a = (plus (times b q) r) & (lessp r b) :  T)}
    from
  [1] {(lessp a b) :  T}
  [2] {a :  N}
  [3] {b :  N}

  with 1 hypotheses

STEP.
```

```
  {(EX q r :  N) (a = (plus (times b q) r) & (lessp r b) :  T)}
    from
 [1] {(EX q r :  N)
          ((diff a b) = (plus (times b q) r) & (lessp r b) :  T)}
  [2] {(lessp a b) = nil}
  [3] {a :  N}
  [4] {b :  N}

  with 1 hypotheses

t
3:(cigIND '{a :  (DD b)} BASIS STEP})    ; Apply CIG induction of  DD.
{*..]- (EX q r :  N) (a = (plus (times b q) r) & (lessp r b) :  T)}
4:display-thm it
it.
  {(EX q r :  N) (a = (plus (times b q) r) & (lessp r b) :  T)}
    from
  [1] {a :  (DD b)}
  [2] {b :  N}

  with 2 hypotheses

t
```

The function `deDP` declares a user defined predicate. A *predicate identifier* must be a symbol whose first character is "`*`" and whose second character is an upper case letter. For example, invoking the command `(deDP *A (x y) {y = x})`, `*A(0 1)` is a formula which is equivalent to `1 = 0`. There are two rules for predicate declaration. `dpI` does folding and `dpE` does unfolding of defined predicate for the conclusion of a proof. The variable list of the declaration must contain all of the free variables of the definition body.

The function `deGP` declares a predicate variable. Such a variable is called a *generic predicate variable* and is always free, i.e., we do not have quantifiers over generic predicate variables. A generic predicate variable is a symbol whose first character is "`^`" and whose second is an upper case letter. For example, invoking `(deGP ^A 2 3)`, `^A` is declared as a predicate variable whose rank is 3 and arity is 2. Since the extraction algorithm needs rank information for any formula, such a declaration of rank is necessary. A rule for generic predicate variables is the instantiation rule `gpE`, which substitute any formula with the same rank to a generic predicate.

Lisp is the metalanguage for **PX**. To define a new function by Lisp, a function `sfdefun` is available. This is the same as ordinary `defun` of Franz Lisp except (i) the function name must not be in use by another function or macro, although one may redefine a function declared by `sfdefun`, (ii) the definition body must be a legal **PX** command. This restriction ensures that such a function declaration does not cause **PX** to crash. Similarly, `sfdefmacro` is a counterpart of `defmacro`. The value assignment is done by `sfsetq`, which is a safe version of `setq`.

7.3. Extractor

The implemented extraction algorithm (*extractor*) is more or less the same as the algorithm presented in 3.2. The name of the extraction algorithm in **PX** is `Extract`. When it applied to a proof with partial variables, then it fails. The function is more or less an implementation of the method described in chapter 3. But it differs from the algorithm of chapter 3 in the following respects:

1. The elimination of partial variables (lemma 3, 3.1) is not done. The extractor applies the algorithm of 3.2 to a given proof and substitutes constants for partial variables in the extracted program. This procedure is correct, for the result is identical to the program extracted from the proof whose partial variables are deleted by the method of lemma 3 of 3.1.
2. The extractor employs a trick by which unnecessary usage of stacks by functions created by CIG recursion is avoided.

The following shows an example of extraction of program from a proof having a partial variable.

```
6.prf1
{]- E (cond (t 1) (t ?1))}
7.(exI (quote {(EX x) (E x)}) it)
{]- (EX x) (E x)}
8.(Extract it)       ; Extract the code.
(cond (t 1) (t dummy1))
```

The following example is a program extraction from the proof shown in a session example in 7.2.6.

```
5:(Extract (vl 3))   ; Extract a program from the proof of event 3.
Function definitions are as follows:

(defrec <DD-0> (a) (b)
        (cond ((lessp a b) (list 0 a))
              (t
               (let!  (((@1:1 @1:2) (<DD-0> (diff a b) b)))
                     (list (add1 @1:1) @1:2)))))
```

```
 The extracted realizer is

 (<DD-0> a b)
```

The function `<DD-0>` with two arguments a, b is defined by CIG recursion extracted from the CIG inductive definition of `DD`, so it is certain that the value of `b` is constant in the course of computation of `(<DD-0> a b)`. When the function `<DD-0>` is declared by `defrec` as above, it can be executed without consuming stacks by pushing either the name or values of `b`. The declaration `(defrec <DD-0> (a) (b) ...)` is the same as the declaration `(deFUN <DD-0> (a b) ...)` for the proof checker. This difference affects only the evaluation of the functions. Users are not allowed to use `defrec`; it is used by the extractor exclusively.

The extracted codes can be run by the **PX** evaluator, and they may also be executed by Franz Lisp. What's more, they may be compiled by the Franz Lisp compiler. The function `export-for-lisp` makes a file including all functions of the present theory. If a file name is an argument of the function, then the function creates a file of the name with an extension `.l`. The file can be compiled by the Franz Lisp compiler. But there is a defect: the codes are not consistent with the standard semantics, for Λ-expressions are compiled. But the compiled functions are consistent with the theory of **PX**. When `export-for-lisp` has an optional argument `not-trick`, it does not do this.

7.4. EKL translator

As mentioned in 7.1, hypotheses may be proved in any other logical system whenever they are sound with in the semantics of **PX**. We present a way to have another proof checker prove hypotheses. We adopt EKL as our target system. EKL is a interactive proof checker developed by Ketonen and Weening 1984, based on a higher order typed language and a term-rewriting system. In EKL formulas are terms of the type of truth values, which is denoted by `truthval`. A term of the type `truthval` is proved when it is rewritten to a constant `true`, so that logical inference can be done by term-rewriting. In typical cases, one can prove a conjecture that is an implication by rewriting its conclusion to `true` step by step, using axioms and its assumption as rewriting rules. One can use also a decision procedure that decides provability of conjectures in a restricted logical system. This procedure is not complete with respect to the complete logical system of EKL, but always terminates. The term-rewriter employed in EKL is so powerful that one can prove conjectures very naturally.

7.4.1. Interpreting LPT of PX in typed language of EKL

To prove hypotheses in EKL, we must translate them and the axiom system of **PX** into EKL. Since EKL is based on typed language, expression, functions, and formulas of **PX** must be translated into terms of appropriate types. EKL has a type named `ground` representing the type of the lowest level objects. We use it as the type that every expression of **PX** has after the translation. That is, all expressions are translated into terms whose types are `ground`. Functions of **PX** must be translated into terms that have the appropriate types according to their arities. For example, *equal* has the type (`ground`$\otimes$`ground`)$\rightarrow$`ground`, where "$\otimes$" and "$\rightarrow$" are EKL primitives that mean the product type and the type of function space, respectively. Some functions of **PX**, e.g., *list*, have infinite numbers of possible arities. EKL has a postfix type operator "`*`" of the union of arbitrary finite products of a given type. So the type of *list* can be represented as `ground`$*\rightarrow$`ground`. Formulas are translated into terms which have the type `truthval` mentioned above.

PX is based on LPT. On the other hand, EKL is based on the usual logic of total terms. So we must translate partial terms into the total terms of EKL. We use the notion *sort* of EKL to realize this. Conceptually, a sort is considered as a predicate over a type, and each variable or constant may have a sort among its attributes. The predicate is considered to be true for any variable or any constant of the sort. We introduce a sort `e`, which corresponds to E of **PX**, and a constant `bottom`, which has the type `ground` and represents the "undefined value" of **PX**. The constant `bottom` does not have the sort `e`. We consider a total expression, i.e., an expression that has a value, of **PX** as an expression that has the sort `e` and others , i.e., expressions that have no value, as expressions whose value is `bottom`. The ordinary equality `=` of the EKL logic of total terms turns out to be Kleene's equality $=$ of **PX**. Thus LPT of **PX** can be translated into the typed language of EKL.

7.4.2. Translating expressions and functions

Each constant and variable of **PX** is translated into a constant and variable of the type `ground`. In EKL, one must declare every constant and every variable before he or she uses them. Constants and total variables are declared with the sort `e` besides its type `ground`. Class variables and class constants are declared with the sort `cl`, which corresponds to *Class* of **PX**. A *total variable (constant) of* EKL is a variable (constant) with the sort `e` or `cl`, and a *partial variable of* EKL is a variable without any sorts. An axiom is included to guarantee that an expression which has the sort `cl` has the sort `e`. So an expression with the sort `cl` is a total expression. The actual system of EKL is *case insensitive*, i.e., it neglects the cases of letters, and it does not permit names of identifiers to include some special characters such as "`*`". But PX is *case sensitive* and permits such special characters among atoms. So we must arrange these to be legal names, distinct from each other. We adopt the method of removing special characters from them and inserting "`_`" immediately before each uppercase letter. If some names are still identical, we add suffixes such as "`_1`" to distinguish them. For example, *atm*, *Atm* and $*Atm$ are converted into `atm`, `_atm` and `_atm_1` respectively. The following are examples of declarations.

```
(decl (px py pz) (type ground))
(decl (x y z) (type ground) (sort e))
(decl (atm) (syntype constant) (type ground) (sort cl)),
```

where (`syntype constant`) means that the identifiers are used as constants. The variables `px`, `py`, `pz` are partial variables, the variables `x`, `y`, `z` are total variables, and the constant `atm` is a total constant. Tuples such as $[e_1, \ldots, e_n]$ are treated as $(e_1, \ldots, e_n)$ in EKL, which has type $\texttt{ground}\otimes\ldots\otimes\texttt{ground}$. EKL identifies (e) with e as **PX** does.

To translate Λ-expressions into EKL, we introduce a notion called Λ-*frame*. A Λ-frame is a context such as $\Lambda(x_1 = *, \ldots, x_n = *)(fn)$ with "holes" denoted by $*$'s. At first, a constant cf whose type is $\texttt{ground}^n\rightarrow\texttt{ground}$ is generated for each Λ-frame f by the following axiom

$$\forall x_1\ \ldots\ x_n\ \texttt{args}.$$
$$\texttt{e}(cf(x_1,\ldots,x_n)) \wedge \texttt{apply}(cf(x_1,\ldots,x_n), \texttt{list}(\texttt{args})) = fn(\texttt{args}),$$

where the variables $x_1, \ldots, x_n$ have the sort `e`, and `args` is declared to have the type $?cf$. In EKL, type identifiers beginning with "?" means *variable types*. A variable type denotes an implicit type, which is automatically assigned to an appropriate type in the context where the variable type occurs. For example, it is $\texttt{ground}\otimes\texttt{ground}$ if fn is *equal*. Note that a variable type is not a polymorphic type variable. All occurrences of the variable type represent an identical type through a proof in EKL. In this setting, each Λ-closure $\Lambda(x_1 = e_1, \ldots, x_n = e_n)(fn)$ is translated to the term $cf(e_1,\ldots,e_n)$, where cf is the constant corresponding to the Λ-frame $\Lambda(x_1 = *, \ldots, x_n = *)(fn)$.

We treat *quote* the same as Λ. We generate one constant corresponding to each atomic expression. We use "`‘`" to generate the constant. "`‘`" is an EKL primitive, which generates a constant ‘a denoting the symbol a itself. If a and b are distinct symbols, EKL automatically regards $\neg$ ‘a=‘b as a predeclared axiom. When the quoted expression is not atomic, we expand it using *pair* and *list*. For example, *quote*$((a\ b\ c))$ and *quote*$((a\ .\ b))$ are converted into `list(‘a,‘b,‘c)` and `pair(‘a,‘b)`, respectively.

EKL has no primitives corresponding to the *let*-quantifier; we must represent this by other EKL primitives. We illustrate this by an example.

$$\begin{aligned}
&\mathit{let}\ (x\ y) = a,\ (x\ 0) = b\ \mathit{in}\ e\\
\Rightarrow\quad &\mathtt{assert}(\mathtt{cddr}(\mathtt{a}) = \mathtt{nil} \wedge \mathtt{car}(\mathtt{b}) = \mathtt{car}(\mathtt{a}) \wedge \mathtt{cadr}(\mathtt{b}) = \mathtt{0} \wedge \mathtt{cddr}(\mathtt{b}) = \mathtt{nil},\\
&\qquad\qquad (\lambda\mathtt{x\ \ y.}\ e)(\mathtt{car}(\mathtt{a}), \mathtt{cadr}(\mathtt{a}))),
\end{aligned}$$

where `assert` is a constant of the type (`truthval`$\otimes$`ground`)$\rightarrow$`ground` with the axiom

$$\forall\mathtt{px.}\ \mathtt{assert}(\mathtt{true}, \mathtt{px}) = \mathtt{px} \wedge \neg\mathtt{e}(\mathtt{assert}(\mathtt{false}, \mathtt{px})),$$

where the variable `px` is a partial variable declared as above.

Each function constant is translated into a constant of functional types. For example, we declare *equal* of **PX** with type (`ground`$\otimes$`ground`)$\rightarrow$`ground`.

We translate **PX**'s *cond* into EKL's constant `cond` of the type

$$(\mathtt{ground}\otimes\mathtt{ground})*\rightarrow\mathtt{ground}.$$

The inference rules for it is translated into the following axiom:

$$\begin{aligned}
&\forall\mathtt{x\ \ px\ \ clauses.}\\
&\qquad \mathtt{cond}() = \mathtt{nil}\\
&\qquad \wedge\ \mathtt{cond}((\mathtt{nil}, \mathtt{px}), \mathtt{clauses}) = \mathtt{cond}(\mathtt{clauses})\\
&\qquad \wedge\ (\neg\mathtt{x} = \mathtt{nil} \supset \mathtt{cond}((\mathtt{x}, \mathtt{px}), \mathtt{clauses}) = \mathtt{px}),
\end{aligned}$$

where the variable `clauses`, `x`, and `px` have type (`ground`$\otimes$`ground`)$*$, `ground`, and `ground` respectively, and `x` has the sort `e`.

EKL supports λ-expressions, so functions represented by λ-expressions in **PX** are straightforwardly representable in EKL. For example, $\lambda(x,y)(x)$ is translated into λ`x y. y`. Whenever EKL executes β-conversion, it checks the sort conditions of the arguments. Since `x` and `y` have the sort `e`, (λ`x y. y`)(e_1, e_2) is not rewritten into e_2 unless $\mathtt{e}(e_1) \wedge \mathtt{e}(e_2)$ holds. This guarantees that β-conversion performed by EKL on translated λ-expressions is the call-by-value β-conversion. We have to add the following axiom for each natural number n to `prove` that β-reduction for the translated λ-expressions is call-by-value:

$$\begin{aligned}
&\forall f_n\ x_1\ \ldots\ x_n\ px_1\ \ldots\ px_n.\\
&\qquad \mathtt{e}((\lambda x_1\ \ldots\ x_n.\ f_n(x_1,\ldots,x_n))(px_1,\ldots,px_n)) \supset \mathtt{e}(px_1) \wedge \ldots \wedge \mathtt{e}(px_n),
\end{aligned}$$

where the types of f_n, x_i and px_i are $\mathtt{ground}^n\rightarrow\mathtt{ground}$, `ground` and `ground`, respectively, and f_n and px_i are partial variable and x_i is a total variable of EKL.

This is consistent with the intended semantics of λ-expression in EKL described in Ketonen and Weening 1983. It does not specify the value of the expression $(\lambda x_1, \ldots, x_n.e)(e_1, \ldots, e_n)$ when one of the values of $e_1, \ldots, e_n$ does not satisfy the sort of the corresponding variable of $x_1, \ldots, x_n$. So we may regard its value as `bottom`, and then the above axiom is satisfied. Since f_n is a partial variable, we may instantiate it by an expression of the form $\lambda x_1, \ldots, x_n.e$ whenever the type of the term `e` is `ground` and the partial variables $px_1, \ldots, px_n$ may be instantiated by any expressions $e_1, \ldots, e_n$ of the type `ground`. Then by the aid of β-reduction in EKL, it turns out to be the axiom

$$\mathtt{e}((\lambda x_1\ \ldots\ x_n.\ e)(e_1, \ldots, e_n)) \supset \mathtt{e}(e_1) \wedge \ldots \wedge (e_n).$$

Application of a function to an expression is denoted as in the concrete syntax of **PX**. This ends the explanation of how expressions and functions of **PX** are translated into EKL. We summarize the translations in the table

PX	EKL
app, *app**	`apply`, `funcall`
pair, *fst*, *snd*, *list*	`cons`, `car`, `cdr`, `list`
atom, *equal*	`atom`, `equal`
suc, *prd*	`add1`, `sub1`
0 , *t* , *nil*, *V*, *N*, *Atm*, *T*	`0 ,t ,nil, _v, _n, _atm, _t`
quote(α)	See above.
$f(e_1, \ldots, e_n)$	$f(e_1, \ldots, e_n)$
$cond(e_1, d_1; \ldots, e_n; d_n)$	$\mathtt{cond}((e_1, d_1), \ldots, (e_n, d_n))$
$\lambda(x_1, \ldots, x_n)(e)$	$\lambda x_1\ \ldots\ x_n.\ e$
$\Lambda(x_1 = e_1, \ldots, x_n = e_n)(e)$	$cf(e_1, \ldots, e_n)$
let $p_1 = e_1, \ldots, p_n = e_n$ *in* e	See above.

7.4.3. Translating formulas

Each formula is translated into a term which has the type `truthval` as mentioned above. Propositional constant $\top$ and $\bot$ are translated into `true` and `false` respectively, which are both primitives of EKL. The predicate E, *Class*, and ":" are translated to have type `ground`$\rightarrow$`truthval`, `ground`$\rightarrow$`truthval` and (`ground`$\otimes$`ground`)$\rightarrow$`truthval`, respectively. We denote them `e`, `cl`, and "`::`", respectively. Predicate "=" of **PX** is translated into "`=`" of EKL. Every propositional connective and quantifier is translated straightforwardly, except $\diamondsuit$, $\rightarrow$, and the ∇-quantifier. The modal operator $\Diamond$ is removed, for $\Diamond$ stands for double negation and EKL follows classical logic.

One possible way of translating $\rightarrow$ is to represent it by means of the other logical symbols as described in 2.3.4. But we introduce a constant `condfml` of the type (`ground`$\otimes$`truthval`)$*\rightarrow$`truthval`. Note that `cond` is the conditional for expressions and `condfml` is the conditional for formulas. We translate the axioms concerning it into EKL as follows:

$$\begin{aligned}&\forall \mathtt{x\ prop\ pclauses}.\\ &\qquad \neg\mathtt{condfml}()\\ &\qquad \wedge\ \mathtt{condfml}((\mathtt{nil},\mathtt{prop}),\mathtt{pclauses}) \equiv \mathtt{cond}(\mathtt{pclauses})\\ &\qquad \wedge\ (\neg \mathtt{x} = \mathtt{nil} \supset \mathtt{condfml}((\mathtt{x},\mathtt{prop}),\mathtt{pclauses}) \equiv \mathtt{prop}),\end{aligned}$$

where `x`, `prop` and `pclauses` are declared with type `ground`, `truthval`, and (`ground`$\otimes$`truthval`)$*$, respectively, and `x` has the sort `e`.

The quantifier ∇ is representable using the existential or universal quantifier as in 2.3.4. But, making effective use of the term-rewriting power of EKL, we instead treat it like the *let*-quantifier, as follows:

$$\begin{aligned}&\nabla(x\ y) = a,\ (x\ 0) = b.\ F\\ &\qquad \Rightarrow\quad \mathtt{cddr}(\mathtt{a}) = \mathtt{nil} \wedge \mathtt{car}(\mathtt{b}) = \mathtt{car}(\mathtt{a}) \wedge \mathtt{cadr}(\mathtt{b}) = 0 \wedge \mathtt{cddr}(\mathtt{b}) = \mathtt{nil}\\ &\qquad\quad\ \wedge\ (\lambda \mathtt{x\ \ y}.\ F)(\mathtt{car}(\mathtt{a}),\mathtt{cadr}(\mathtt{a})).\end{aligned}$$

Note that `x` and `y` must be total variables, i.e., they must have been declared to have the sort `e`. To justify this translation, we must add the following axiom as in the case of call-by-value λ-expression.

$$\begin{aligned}&\forall phi_n\ x_1 \ldots x_n\ px_1 \ldots px_n.\\ &\qquad (\lambda x_1 \ldots x_n.\ phi_n(x_1,\ldots,x_n))(px_1,\ldots,px_n) \supset \mathtt{e}(px_1) \wedge \ldots \wedge \mathtt{e}(px_n),\end{aligned}$$

where phi_n is declared to have type $\mathtt{ground}^n\rightarrow\mathtt{truthval}$ and any other variables are the same as the case of call-by-value λ-expressions.

Sorts provide a simple method of relativizing quantifications. For example, suppose that variables `x` and `y` are declared to have the sorts `S` and the universal sort (i.e., the sort λ`y. true`), respectively. Then $\forall$`x. P(x)` means $\forall$`y. S(y)` $\supset$ `P(y)`. Such a relativization is also available for $\exists$. The totalness condition of $(\forall E)$ and $(\exists I)$ in **PX** are represented as conditions concerning sort in EKL. This is not only syntactically simple, but also good for preserving the term-rewriting power of EKL. Translation of formulas is summarized in table 2.

Table 2

PX	EKL
$\top,\ \bot$	`true, false`
$[e_1, \ldots, e_n] : e$	$(e_1, \ldots, e_n)::e$
$e_1 = e_2$	$e_1{=}e_2$
$E(e)$	$\mathtt{e}(e)$
$Class(e)$	$\mathtt{cl}(e)$
$F_1 \wedge \ldots \wedge F_n$	$F_1 \wedge \ldots \wedge F_n$
$F_1 \vee \ldots \vee F_n$	$F_1 \vee \ldots \vee F_n$
$F_1 \supset F_2$	$F_1 \supset F_2$
$F_1 \supset\subset F_2$	$F_1 \equiv F_2$
$\neg F$	$\neg F$
$\Diamond F$	F
$e_1 \to F_1; \ldots ; e_n \to F_n$	$\mathtt{condfml}((e_1, F_1), \ldots, (e_n, F_n))$
$\nabla p_1 = e_1, \ldots, p_n = e_n.F$	See above.
$\forall [x_1, \ldots, x_l] : e_1, \ldots, [y_1, \ldots, y_m] : e_n.F$	$\forall x_1\ \ldots\ x_l\ \ldots\ y_1\ \ldots\ y_m.$ $(x_1, \ldots, x_l)::e_1 \wedge \ldots \wedge (y_1, \ldots, y_m)::e_n \supset F$
$\exists [x_1, \ldots, x_l] : e_1, \ldots, [y_1, \ldots, y_m] : e_n.F$	$\exists x_1\ \ldots\ x_l\ \ldots\ y_1\ \ldots\ y_m.$ $(x_1, \ldots, x_l)::e_1 \wedge \ldots \wedge (y_1, \ldots, y_m)::e_n \wedge F$

7.4.4. Translating axiom system

Now we explain how the axiom system of **PX** is translated into EKL. The almost all axioms and inference rules of **PX** need not be translated into EKL, because EKL has a set of usual axioms for higher order logic.

We generate one declaration and three axioms for each CIG inductive definition. We declare the defined symbol to have the type appropriate to the number of its parameters. The first axiom indicates that the defined symbol has the sort `cl` and the second indicates that the class is a fixed point of the defining equation. The last one is the schema of the CIG induction. The following is the example of the case of *List* in 2.4.1. At first, *List* is declared with type `ground`$\to$`ground`. Suppose that `_a`, `x` and `phi` are declared to have type `ground`, `ground` and (`ground`$\otimes$`ground`)$\to$`truthval`, respectively and `_a` has the sort `cl`.

$$\forall \mathtt{\_a}.\ \mathtt{cl(\_list(\_a))}$$

$$\begin{aligned}&\forall \mathtt{\_a\ x}.\\ &\quad \mathtt{x} : \mathtt{\_list(\_a)}\\ &\qquad\qquad \equiv \mathtt{condfml}((\mathtt{atom}(\mathtt{x}), \mathtt{x} = \mathtt{nil}), (\mathtt{t}, \mathtt{fst}(\mathtt{x}) : \mathtt{\_a} \wedge \mathtt{snd}(\mathtt{x}) : \mathtt{\_list(\_a)}))\end{aligned}$$

$$\begin{aligned}
&\forall \mathtt{phi\ \_a}.\\
&\quad (\forall \mathtt{x}.\ (\mathtt{condfml}((\mathtt{atom}(\mathtt{x}), \mathtt{x} = \mathtt{nil}),\\
&\qquad\qquad (\mathtt{t}, \mathtt{fst}(\mathtt{x}) : \mathtt{\_a} \wedge \mathtt{snd}(\mathtt{x}) : \mathtt{\_list}(\mathtt{\_a}) \wedge \mathtt{phi}(\mathtt{snd}(\mathtt{x}))))\\
&\qquad \supset \mathtt{phi}(\mathtt{x}))\\
&\quad \supset (\forall \mathtt{x}.\ \mathtt{x} : \mathtt{\_list}(\mathtt{\_a}) \supset \mathtt{phi}(\mathtt{x})),
\end{aligned}$$

where `condfml` is the constant corresponding to $\rightarrow$ and `_list` corresponds to *List*.

Each function or individual constant defined by a user of **PX** is declared with an appropriate type and the axiom indicating that the function or the constant is a fixed point of the equation given in the definition. Axioms for the modal operator $\diamondsuit$ are not necessary, since it is removed when formulas are translated. The totality and strictness properties of functions are represented as sorts. For example, we declare predicate constants `total2` and `strict2` with type ((`ground`$\otimes$`ground`)$\rightarrow$`ground`)$\rightarrow$`truthval`, then we declare total and strict functions with two arguments such as `equal` to have the sorts `total2` and `strict2`, adding the axioms

$$\begin{aligned}
&\forall \mathtt{tf\ x\ y}.\ \mathtt{e}(\mathtt{tf}(\mathtt{x}, \mathtt{y}))\\
&\forall \mathtt{sf\ px\ py}.\ \mathtt{e}(\mathtt{sf}(\mathtt{px}, \mathtt{py})) \supset \mathtt{e}(\mathtt{px}) \wedge \mathtt{e}(\mathtt{py}),
\end{aligned}$$

where `tf`, `sf`, `x` and `y` have the sort `total2`, `strict2`, `e` and `e`, respectively, and `px` and `py` have the universal sort. The axioms for quoted constants are automatically obtained by the EKL system, for we adopt the method presented in 7.4.2.

7.4.5. How to call EKL from PX

The following illustrates how to call EKL from **PX**.

```
1:  eklI {x = z} {x = y} {y = z}
Sending hypothesis.

>Wait.
127.  (show hyp hyp_c)

;labels:  PRF HYP_C
124.  (DEFAX HYP_C |HYP_C(X,Z) IFF X = Z|)

;labels:  PRF HYP
126.  (DEFAX HYP |HYP IFF (X = Y&Y = Z IMP HYP_C(X,Z))|)

128.  (trw hyp (open hyp hyp_c))
```

```
;HYP

129.  (qed)
;Please type Control-D
Bye.
{..]- x = z}
```

We call EKL by the function `eklI`. To prove a hypothesis $\{F_1, \ldots, F_n\} \Rightarrow F$, we invoke a command (`eklI` F $F_1 \ldots F_n$). Then the EKL translator translates the hypothesis, calls EKL, and sends the necessary stuff to EKL. The first line above is a line of **PX** and the command on it invokes EKL. The lines 127 to 129 are lines of EKL. The command in line 127 displays the hypothesis labeled as `hyp` and its conclusion labeled as `hyp_c`. Since the conclusion of the hypothesis has two free variables `x`, `y`, it is `hyp_c(x,y)` instead of `hyp_c`. The command of the line 127 proves the hypothesis, and the command `(qed)` of line 129 checks if the hypothesis is proved. To exit from EKL, we type in Control-D. EKL returns a signal of the completion of the proof to **PX** and `eklI` makes a proof of the hypothesis as the result of the command of the line 1 of **PX**. When you exit from EKL without completing the proof, then EKL returns a signal indicating failure of the proof and `eklI` fails.

8 Conclusion

8.1. What was done?

The possibility of program extraction has been widely known in constructive mathematics and computer science as the “proofs as programs” principle. In spite of some interesting work, it has not been made so clear how feasible programs are gotten from constructive proofs. When it came to the efficiency of produced programs, many people were pessimistic. We built a computer system which suggests that there is hope to write efficient realistic programs by constructive proofs. Using a new viewpoint to the induction principle and the termination problem, we presented a general method to extract efficient Lisp programs.

We presented a type free logical system which serves as a foundation for the rigorous development of functional programs. The flexibility of the type free approach was shown through program extraction and interpretation of types.

Apart from the theoretical aspects, we presented a proof checker with an extensible set of inference macros which resembles natural language, and can be printed by TEX.

8.2. What is missed? What should be done?

We proved that all recursive functions can be extracted from **PX** by adding all true rank 0 formulas. But this can be done only for first order functions. It is unclear to what extent higher order functions can be extracted from constructive proofs. Extraction of efficient higher order programs and satisfactory characterization of higher order programs remain unsolved. Difficulty of characterization of higher order programs reflects in the result in 2.4.3. This is one of the most interesting subjects in theoretical studies in our direction.

The interactive interface of **PX** is still in a very primitive stage, when it is compared to Nuprl. It seems that there is no theoretical difficulty in adding tactic-like features to **PX**. But it will take great effort. Although tactic-like features and Nuprl-like interface help proof development to a great extent, it deos not useful to present completed proofs to the others. Our inference macros and TEX translator will help. But there is no logical relationship internal proofs in **PX** and proofs in TEX. Something which bridges them is expected. It would be useful to build a computerized textbook of mathematics.

The optimizers used in the extraction algorithm are still not satisfactory, although they work well in most cases. There is a theoretical drawback to optimizations of codes in the type free approach (see appendix C). This needs further study. Aside from this drawback, further optimizations will be possible by using information contained in source proofs. For example, an extractor can decide types of expressions in extracted programs, looking up source proofs. This will enable the extractor to add type declarations for optimizations. It is expected to increase efficiency of compiled codes to a great extent.

The treatment of function closure (Λ-expression) is not quite elegant. This is mainly because that we obey syntax and semantics of Lisp. It made us use functional argument with free variables and intensional equality in comparing functions. This could be solved by replacing Lisp by a more idealistic language, e.g., CAM (Cousineau, Curien, and Mauny 1985), and/or the introduction of a class of function values and extensional equality between functions. Then decidability of equality cannot be retained. But, this is not serious as T_0 does not have it. The reason why we included the decidability is that we intended the universe V as the domain of Lisp.

A Comparison of px-realizability and other interpretations

There are many possible ways to extract the computational content of a constructive proof. In this appendix, we compare them with **px**-realizability. The interpretations we examine are **r**-realizability, **q**-realizability, Grayson's **g**-realizability Gödel interpretation, modified realizability, and the normalization method.

Ordinary realizability, which we refers to as **r**-realizability, is more transparent than **px**-realizability as a foundation of "formulas as types". Its defect is that the double negation shift (the axiom ($\Diamond 2$)), from which proposition 3 of 2.3.5 follows, does not hold in the interpretation. Furthermore, even if f is a realizer of $\forall x.\exists y.A$, it does not guarantee that $\forall x.\nabla y = f(x).A$ holds *classically*. For example, Church's axiom (see Troelstra 1973) is realized by the interpretation, but it does not hold classically. This conflicts with our point of view that a specification is given by the $\forall\exists$-formula under the interpretation of ordinary logic.

Grayson's realizability, which we refer to as **g**-realizability, is one of motivations of **px**-realizability. It is given by modifying clause 1 of definition 1 of 3.1 of **px**-realizability as follows:

1. A is an atomic formula. Then a **x** A is A.

It is a variant of **q**-realizability and is essentially the semantics of the "logical space" obtained by gluing (in the sense of Artin) the logical space obtained by **r**-realizability and the logical space of sets and functions (Johnstone 1977). Grayson's realizability is *not* equivalent to traditional **q**-realizability; e.g.,

$$\neg\forall x(\exists y.T(x,x,y) \vee \neg\exists y.T(x,x,y))$$

is **q**-realized but not **g**-realized. The relation between **q**-realizability and **g**-realizability is that a **g**-realizes A iff a **q**-realizes A and A holds. (The relation of the realizability in Hayashi 1983 and **px**-realizability is the same.) When we allow only valid sequents, these **q**-realizability and **g**-realizability are essentially the same.

For a rank 0 formula A, if A holds, then A is **q**-realizable. Hence A holds iff A is **g**-realizable for any rank 0 formula A. This also holds for **px**-realizability. The only difference between **px**-realizability and **g**-realizability is the difference of realizers of rank 0 formulas. Realizers of rank 0 formulas are deliberately restricted to *nil* in **px**-realizability. This accords with the fact that the type $I(A,x,y)$

of Martin-Löf's type theory has at most one element. This also leads the simplification of types represented by formulas in refined **px**-realizability such as $\mathbf{x}(A \times B) \cong \mathbf{x}(B)$ for the rank 0 formula A. The favorable aspect of **g**-realizability and **px**-realizability is that whenever a formula is realizable, then it holds classically. This does not hold for **q**-realizability, e.g., Church's axiom holds under it.

It is noteworthy that the realization of a formula whose is not rank 0 cannot be rank 0 in **g**-realizability and **px**-realizability. On the other hand, all realization formulas of the traditional realizabilities are rank 0. The property "if A is realizable, then A holds" contradicts to the property "realizations are rank 0 formulas". Traditional realizabilities include the latter property, while we include the former.

Modified realizability is known as a "typed realizability. But usage of type is *not* the essential difference from the other realizability. The difference is the interpretation of implication. If A is a rank 0 formula in our sense, then the interpretation of $A \supset B$ in modified realizability is

$$a \ \mathbf{mr}\ A \supset B \equiv A \supset a \ \mathbf{mr}\ B.$$

Since A does not have any computational information, a realizer for A is not necessary for the computation of a realizer of B. This appears to be better than our realizability; but the above modification of the interpretation of implication contradicts LPT. For example, the following is a theorem of LPT, but it is not realized by modified realizability:

$$E(e) \supset \exists x.(e = x).$$

If e_1 is a realizer of this, then e_1 should have a value, and whenever $E(e)$ holds it must coincide the value of e. Let T be Kleene's T-predicate (Kleene 1953). If we take $\mu y.T(x, x, y)$ as e, then the existence of such e_1 for e contradicts the undecidability of the termination problem. If all terms have value, such a difficulty does not appear. In the usual typed logical theories, all terms have value. This is the reason why modified realizability fits typed theories rather than untyped theories like **PX**. Since programs need not have values, it seems that ordinary realizability is more appropriate as a program extraction method than modified realizability. Modified realizability is not appropriate as an extraction method for **PX**, which is based on the logic of partial terms.

Note that the above argument depends on the assumption that a realizer is a value or an expression whose computation terminates. We think this assumption is fair. Even if it were possible to define a realizability so that a realizer need not have a value, there would be no guarantee that $f(x)$ terminates on arbitrary input x, for a realizer f of $\forall x.\exists y.A$. One might insist that $f(x)$ has a value, thinking nontermination as a value as in models of λ-calculus or in the logic of partial existence. But when y is intended to be a concrete datum like an integer, such an argument seems to be unacceptable.

The interpretation that was first used to extract programs from proofs is the Gödel interpretation (Goto 1979, 1979a, Sato 1979). But it is not quite adequate as a tool for program extraction. The interpretation of the implication is rather intricate, and the form of the interpretation is $\exists\vec{x}\forall\vec{y}.A$, where A does not contain quantifiers. So it is hard to arrive at the interpretation of a formula with complicated implication and quantifiers, as was pointed out by Goto 1979a. This implies that it is difficult to represent an intended specification by a formula. On the other hand, it is easy to write an intended specification by a formula through a realizability as we saw in chapter 4.

Beeson 1978 pointed out that the Gödel interpretation fits neither in the natural deduction nor the logic of partial terms. The most difficult logical rule to verify in the Gödel interpretation is $A \supset A \wedge A$ or $A \supset B, A \supset C/A \supset B \wedge C$ (see Beeson 1978, Troelstra 1973). This is "sharing of an assumption by different proofs" and corresponds to "sharing of an input by different programs". Such a situation quite often appears in natural deduction proofs and programming. Let $\exists u^F \forall v^F.F_D$ be the Gödel interpretation of F. (In the original Gödel interpretation, u^F and v^F are sequents of variables, but for simplicity we think they are single variables of appropriate Cartesian types.) Then the Gödel interpretation of a sequent $\{A\} \Rightarrow B$ would be

$$\{A_D(u^A, v^A(u^A, v^B))\} \Rightarrow B_D(u^B(u^A), v^B).$$

A term $\langle v^A, u^B\rangle$ satisfies this interpretation if this sequent holds. Let $\langle v_1^A, u^{B_1}\rangle$ and $\langle v_2^A, u^{B_2}\rangle$ be terms satisfying the upper sequents of the rule

$$\frac{\{A\} \Rightarrow B_1 \qquad \{A\} \Rightarrow B_2}{\{A\} \Rightarrow B_1 \wedge B_2}.$$

Although both v_1^A and v_2^A give the existential information of A in the sequent, they may differ. For example, if A is a universal formula and is instantiated by two different terms to derive B_1 and B_2, then they differ. Let us denote $B_1 \wedge B_2$ by C. Then v^C is normally

$$v^C(u^A, v^A(u^A, v^C)) = \begin{cases} v_2^A(pr_1(v^C), pr_2(v^C)) & \neg A_D(u^A, v_2^A(pr_1(v^C), pr_2(v^C))) \\ v_1^A(pr_1(v^C), pr_2(v^C)) & \text{otherwise} \end{cases}$$

For any inference rules that have more than one upper sequent, we have to do the same thing. Since many inference rules of natural deduction have more than one upper sequent, this complicates the interpretation of natural deduction. Furthermore, Beeson pointed out that the quantifier-free formula A_D *must be decidable* in order to carry out the above definition, but quantifier-free formulas of LPT may not be decidable. Beeson 1978 used a Diller-Nahm variant of the Gödel interpretation for T_0. His interpretation of the implication is also complicated, and the problem of "sharing of assumption" still remains. The Gödel interpretation is therefore not suitable for program extraction, especially not for "programming via proofs". Recent researches in liner logic show that the Gödel interpretation is suitable to linear logic. Linear logic is a logic without the sharing of assumptions. Liner logic and the Gödel interpretation might be useful for program extraction of programs for which sharing of variables is dangerous, e.g., programming with many processes.

The normalization method is not exactly program extraction, for a proof itself is executed. But when λ-terms like codes are extracted from proofs as was done by Goad 1980, we may regard it as program extraction for which the execution of extracted codes is done by β-reduction. Then, as was pointed out by Goad 1980, the extraction method is almost the same as modified realizability. The advantage is that logical information kept by proofs may be used as optimization of computation, as was also shown by Goad. The disadvantage is that the execution method is fixed. It would be possible to "compile" extracted codes to other algorithmic languages. In a sense, our realizability method is one of such possible compilation methods. On the other hand, it would be possible to compile proofs through the realizability to codes of an appropriate language so that the compiled codes keep the logical information of the source proofs to do optimizations as was done by the proof interpreter of Goad 1980. It seems the relation between the realizability and normalization methods is similar to the relation between a compiler and interpreter. So it is difficult to say which is better.

B Extraction of a tautology checker of propositional logic

In this appendix, we give an example of programming via proofs with **PX**. We define a formal system PL of propositional logic and prove its completeness theorem. From the proof of the completeness theorem, we extract a program that returns a proof of a given sequent, when it is provable, and returns a refutation, i.e., a valuation under which the sequent is false, when it is not provable. The algorithm adopted by the program is the Wang algorithm, i.e., the algorithm of the program of McCarthy 1965, VIII. Boyer and Moore 1979 have verified a tautology checker of propositional logic. Their program checks if a formula is a tautology, but does not return a proof. Shankar 1985 has verified essentially the same theorem as ours by means of Boyer-Moore theorem prover. But the algorithms employed are completely different from ours.

We give an informal description of the system of PL before giving its formal development. PL has only one logical connective, implication "$\supset$", and one propositional constant, falsehood "$\perp$". A formula of PL is constructed by these from propositional variables. PL is a sequent calculus. A sequent is a form $g_1 \Rightarrow g_2$, where g_1 and g_2 are lists of formula. The inference rules of PL are

$$\text{(falsehood)} \quad g_1 \Rightarrow g_2 \quad (g_1 \text{ contains } \perp)$$

$$\text{(intersect)} \quad g_1 \Rightarrow g_2 \quad (g_1 \text{ and } g_2 \text{ have a common element})$$

$$\text{(permutation)} \quad \frac{g_1 \Rightarrow g_2}{g_3 \Rightarrow g_4} \quad (g_3 \text{ and } g_4 \text{ are permutations of } g_1 \text{ and } g_2)$$

$$\text{(impI)} \quad \frac{(A \;.\; g_1) \Rightarrow (B \;.\; g_2)}{g_1 \Rightarrow (A \supset B \;.\; g_2)} \qquad \text{(impE)} \quad \frac{g_1 \Rightarrow (A \;.\; g_2) \quad (B \;.\; g_1) \Rightarrow g_2}{(A \supset B \;.\; g_1) \Rightarrow g_2}$$

A valuation are a function from propositional variables to Boolean values. A refutation of sequent $g_1 \Rightarrow g_2$ is a valuation such that all formulas of g_1 are true and all formulas of g_2 are false under it. What we have proved with **PX** is that for each sequent, its proof exists or its refutation exists.

Next we explain how we represented PL in **PX**. Propositional variables are represented by atoms, except *nil*. *nil* represents $\perp$. A formula $A \supset B$ is represented by a list $(A\ B)$. A sequent $g_1 \Rightarrow g_2$ is also represented by a list $(g_1\ g_2)$. A proof represented by a list such that the first element is the name of the last inference rule, the second element is the lower sequent of the rule, and the remaining elements are the immediate subproofs. Truth values are represented by t and nil. A valuation is a function closure whose range is $\{t, nil\}$ such that nil is mapped to nil.

The formal development of these representation is

```
; x is supposed to be a list
(deFUN existNonAtom (x)
  (cond ((null x) nil)
        ((atom (car x)) (existNonAtom (cdr x)))
        (t t)))

; Assume x contains a non-atom.
; Then this returns a triple (a b c) s.t.  a*<b>*c=x and
; a is a list of atoms, b is a non-atom and c is the rest.
(deFUN divide (x)
 (cond ((atom (car x))
        (let!  (((a b c) (divide (cdr x))))
               (list (cons (car x) a) b c)))
       (t
        (list nil (car x) (cdr x)))))

; Formulas of PL. (a b) is interpreted as a -> b
(deCIG {x :  Fm}
  ((atom x) {TRUE})
  (t {(LET ((a b) x)) (a :  Fm & b :  Fm)}))

; Lists of formulas
(deECA {x :  FmL} {x :  (List Fm)})

; Formula lists containing a non-atomic formula
(deECA {x :  NonTrivFmL} {x :  FmL & (existNonAtom x) :  T})

(deECA {x :  TrivFmL} {x :  FmL & - x :  NonTrivFmL})

; Function space
(deECA {f :  (Func X Y)} {(UN x :  X)((funcall f x) :  Y)})

; Valuation
(deECA {v :  Val} (in (Func Atm Bool)) {(funcall v nil) = nil})

; Truthvalue of x:Fm under the valuation r:Val
(deFUN truthfml (x r)
  (cond ((atom x) (funcall r x))
        (t (or (not (truthfml (car x) r))
         (truthfml (cadr x) r)))))

; All formulas in x:FmL are true under r:Val
(deFUN truthand (x r)
  (cond ((dtpr x) (and (truthfml (car x) r) (truthand (cdr x) r)))
        (t t)))

; There is a true formula in x:FmL under r:Val
```

```
(deFUN truthor (x r)
  (cond ((dtpr x) (or (truthfml (car x) r) (truthor (cdr x) r)))
        (t nil)))

; Fetch subproofs
(deFUN subprf1 (x) (caddr x))

(deFUN subprf2 (x) (cadddr x))

; g1 is a permutation of g2
(deECA {g1 g2 : Perm}
            {(UN x)((member x g1) : T <-> (member x g2) : T)})

; Definition of proofs of PL.
(deCIG {x : Prf} (in Dp)

  ((equal (car x) 'permutation)
   {(subprf1 x) : Prf &
    (LET (('permutation (g1 g2) (n (g3 g4) d)) x))
       (g1 : FmL & g2 : FmL & g3 : FmL & g4 : FmL
                 & g1 g3 : Perm & g2 g4 : Perm)})

  ((equal (car x) 'intersect)
   {(LET ((a b) (cadr x)))
          (a : FmL & b : FmL &
            $((EX x)((member x a) : T & (member x b) : T)))})

  ((equal (car x) 'falsehood)
   {(LET ((a b) (cadr x)))(a : FmL & b : FmL & (member nil a) : T)})

  ((equal (car x) 'impI)
   {(LET ((a b) (cadr x)))
     ((subprf1 x) : Prf &
      (LET (('impI (g1 ((f1 f2) . g2)) (n ((f1 . g1)(f2 . g2)) d)) x))
                         (TRUE))})

  ((equal (car x) 'impE)
   {(LET ((a b) (cadr x)))
          ((subprf1 x) : Prf
           & (subprf2 x) : Prf
            & (LET (('impE (((f1 f2) . c) g2)
                        (n1 (c (f1 . g2)) d1)
                           (n2 ((f2 . c) g2) d2)) x))
                   (TRUE))}))

; This CIG represents the control structure of the target program.
(deCIG

  ; top level of the program
  ({g1 g2 : Right} (in (Cartesian FmL FmL))
      ((and g2 (existNonAtom g2)) {g1 g2 : PermR})
      (t {g1 g2 : Left}))
  ; g2 = nil or all elements of g2 are atoms.
  ({g1 g2 : Left} (in (Cartesian FmL TrivFmL))
       ((and g1 (existNonAtom g1)) {g1 g2 : PermL})
       (t {TRUE}))
  ({g1 g2 : RuleR} (in (Cartesian FmL FmL))
       (t {(LET (((a b) . c) g2)) ((cons a g1) (cons b c) : Right)}))

  ({g1 g2 : RuleL} (in (Cartesian FmL FmL))
       (t {(LET (((a b) . c) g1))
           (c (cons a g2) : Right & (cons b c) g2 : Right)}))

  ({g1 g2 : PermR} (in (Cartesian FmL NonTrivFmL))
       (t {(LET ((a b c) (divide g2)))
             (g1 (append (list b) c a) : RuleR)}))

  ({g1 g2 : PermL} (in (Cartesian NonTrivFmL FmL))
       (t {(LET ((a b c) (divide g1)))
             ((append (list b) c a) g2 : RuleL)})))

; The conclusion of a proof x
(deFUN con (x) (cadr x))
```

The following are programs which build a proof of the completeness theorem of PL. The proof of the completeness theorem is assigned to the variable `Wang`.

```
; User-defined inference macros used in the proof.

(sfdefmacro LEMMA: (name proof)
   `(progn (deprf ,name ,proof) (display-thm ,name)))
(sfdefmacro THEOREM: (name proof)
   `(progn (deprf ,name ,proof) (display-thm ,name)))
(sfdefmacro Under_the_assumptions arg `(Suppose . ,arg))
(sfdefmacro Under_the_assumption arg `(Suppose . ,arg))
(sfdefmacro by_apply_induction_of arg `(Apply_induction_of . ,arg))
(sfdefmacro Make_assumption arg `(assume . ,arg))
(sfdefmacro the_left_conjunct_of (x) (sflist 'conjE x 1))
(sfdefmacro the_right_conjunct_of (x) (sflist 'conjE x 2))


(LEMMA: Lemma1

 (We_prove (CON)

   {(UN g1 g2 : (List V))
      ($(EX x)((member x g1) : T & (member x g2) : T)
        +
       - (EX x)((member x g1) : T & (member x g2) : T))}

 ;PROOF
```

```
(@
 (Under_the_assumptions

   (Asp1) {g1 :  (List V)}
   (Asp2) {g2 :  (List V)}

  (We_prove

    (CON1) {$(EX x)((member x g1) :  T & (member x g2) :  T)
             +
            - (EX x)((member x g1) :  T & (member x g2) :  T)}

   (by_apply_induction_of {g1 :  (List V)} to ;INDUCTION STEP and
     ;BASE CASE

   ;INDUCTION STEP

   (Suppose

      (INDHYP1) {(dtpr g1) :  T}
      (INDHYP2) {(cdr g1) :  (List V)}
      (INDHYP3) {$(EX x)((member x (cdr g1)) :  T & (member x g2) :  T)
                   +
                 -(EX x)((member x (cdr g1)) :  T & (member x g2) :  T)}

   (We_prove CON1 (by_cases (A) or (B) of INDHYP3

    In_the_case (A)

     (We_see CON1 (since (obviously

             {$(EX x)((member x g1) :  T & (member x g2) :  T)})))

    In_the_case (B)

    (We_prove CON1 (by_cases (B1) otherwise (B2) of (the_fact
      {(OR (member (car g1) g2) t)})

      In_the_case (B1)

     (We_see CON1 (since (obviously

             {$(EX x)((member x g1) :  T & (member x g2) :  T)})))

      In_the_case (B2)

     (We_see CON1 (since (obviously

             {-(EX x)((member x g1) :  T & (member x g2) :  T)}))))))))

   ;BASE CASE

   (Suppose
      (INDHYP4) {g1 = nil}

     (We_see CON1 (since (obviously

              {-(EX x)((member x g1) :  T & (member x g2) :  T)})))))))

 (Hence CON))))


; Shorthands

(sfdefun Provable fexpr (l)
  (let (((x y) l))
    `{(EX p :  Prf) (LET ((h1 h2) (con p)))(h1 = ,x & h2 = ,y)}))

(sfdefun Refutable fexpr (l)
  (let (((x y) l))
    `{(EX r :  Val)((truthand ,x r) :  T & (truthor ,y r) = nil)}))

(Let CON be `{_(Provable g1 g2) + _(Refutable g1 g2)})



; Proof of Completeness Theorem

(LEMMA: Right1 (Make_assumption CON))

(LEMMA: Right2 (Make_assumption CON))

(LEMMA: Left1 (Make_assumption CON))

(LEMMA: Left2

 (Under_the_assumptions

   (INDHYP0) {g1 g2 :  (Cartesian FmL TrivFmL)}

   (INDHYP1) {(and g1 (existNonAtom g1)) = nil}

  (We_prove CON (by_cases (A) otherwise (B) of (the_fact

    {(OR (member nil g1) t)})

  In_the_case (A)

   (We_see CON (since (obviously (Provable g1 g2)
    regarding (list 'falsehood (list g1 g2) as p))))

  In_the_case (B) ; (member nil g1) = nil

    (@
     (Obviously {g1 :  (List V)})
     (Obviously {g2 :  (List V)})
     (We_instantiate g1 and g2 of Lemma1 by g1 and g2)
     (We_prove CON (by_cases (B1) or (B2) of the_previous_fact

      In_the_case (B1)

      (We_see CON (since (obviously (Provable g1 g2)
       regarding (list 'intersection (list g1 g2)) as p)))

      In_the_case (B2)

      (We_see CON (since (obviously (Refutable g1 g2)
       regarding

        (Lambda ((g1 g1)) (lambda (a) (cond ((member a g1) t))))

       as r))))))))))


(LEMMA: RuleR

  (Suppose

    (INDHYP0) {g1 g2 :  (Cartesian FmL FmL)}

   (We_assume

    (INDHYP1:1) ((a b) .  c) matches g2
    (INDHYP1:2) g2 exists
    (INDHYP1:3) {(EX p :  Prf)
                   (LET ((h1 h2) (con p)))
                     (h1 = (cons a g1) & h2 = (cons b c))
                  +
                (EX r :  Val)
                   ((truthand (cons a g1) r) :  T
                     & (truthor (cons b c) r) = nil)}

  (We_prove CON (by_cases (A) or (B) of INDHYP1:3

    In_the_case (A)

     (Since A (we_may_assume there_exists p1 such_that

      (A1) {p1 :  Prf}
      (A2) {(LET ((h1 h2) (con p1)))
            (h1 = (cons a g1) & h2 = (cons b c))}

     (We_see CON (since (obviously (Provable g1 g2)
      regarding (list 'impI (list g1 g2) p1) as p)))))

    In_the_case (B)

     (Since B (we_may_assume there_exists r1 such_that
      (B1) {r1 :  Val}
      (B2) {(truthand (cons a g1) r1) :  T
            & (truthor (cons b c) r1) = nil}

     (@
       (Obviously {(truthand g1 r1) :  T & (truthor g2 r1) = nil})
       (We_see (Refutable g1 g2))
       (Hence CON))))))))

(LEMMA: RuleL

  (Suppose

    (INDHYP0) {g1 g2 :  (Cartesian FmL FmL)}

   (We_assume

    (INDHYP1:1) ((a b) .  c) matches g1
    (INDHYP1:2) g1 exists
    (INDHYP1:3)
      {(((EX p :  Prf)
           (LET ((h1 h2) (con p)))(h1 = c & h2 = (cons a g2))
                 +
         (EX r :  Val)
            ((truthand c r) :  T & (truthor (cons a g2) r) = nil))

          &

        ((EX p :  Prf)
          (LET ((h1 h2) (con p)))(h1 = (cons b c) & h2 = g2)
                  +
         (EX r :  Val)
            ((truthand (cons b c) r) :  T & (truthor g2 r) = nil)))}

   (We_prove CON (by_cases (A) or (B) of (the_left_conjunct_of INDHYP1:3)

     In_the_case (A)

      (We_prove CON (by_cases (C) or (D) of (the_right_conjunct_of
        INDHYP1:3)

     In_the_case (C)

         (Since A (we_may_assume there_exists p1 such_that

         (A1) {p1 :  Prf}
         (A2) {(LET ((h1 h2) (con p1)))(h1 = c & h2 = (cons a g2))}

          (Since C (we_may_assume there_exists p2 such_that

           (C1) {p2 :  Prf}
           (C2) {(LET ((h1 h2) (con p2)))(h1 = (cons b c) & h2 = g2)}
          (We_see CON (since (obviously (Provable g1 g2)
           regarding (list 'impE (list g1 g2) p1 p2) as p)))))))

     In_the_case (D)

         (Since D (we_may_assume there_exists r1 such_that

         (D1) {r1 :  Val}
         (D2) {(truthand (cons b c) r1) :  T & (truthor g2 r1) = nil}

          (@
            (Obviously {(truthand g1 r1) :  T & (truthor g2 r1) = nil})
            (We_see (Refutable g1 g2))
            (Hence CON))))))

     In_the_case (B)

      (Since B (we_may_assume there_exists r2 such_that

        (B1) {r2 :  Val}
        (B2) {(truthand c r2) :  T & (truthor (cons a g2) r2) = nil}

      (@
        (Obviously {(truthand g1 r2) :  T & (truthor g2 r2) = nil})
        (We_see (Refutable g1 g2) (by B1))
        (Hence CON)))))))))


(LEMMA: PermR

  (Suppose

    (INDHYP0) {g1 g2 :  (Cartesian FmL NonTrivFmL)}

   (We_assume

    (INDHYP1:1) (a b c) matches (divide g2)
    (INDHYP1:2) (divide g2) exists
    (INDHYP1:3) {(EX p :  Prf)
                   (LET ((h1 h2) (con p)))
                          (h1 = g1 & h2 = (append (list b) c a))
                  +
                (EX r :  Val)
                   ((truthand g1 r) :  T
                       & (truthor (append (list b) c a) r) = nil)}

   (We_prove CON (by_cases (A) or (B) of INDHYP1:3

     In_the_case (A)

      (Since A (we_may_assume there_exists p1 such_that
        (A1) {p1 :  Prf}
        (A2) {(LET ((h1 h2) (con p1)))
                 (h1 = g1 & h2 = (append (list b) c a))}

     (We_see CON (since (obviously (Provable g1 g2)
        regarding (list 'PermR (list g1 g2) p1) as p)))))

     In_the_case (B)

      (Since B (we_may_assume there_exists r1 such_that

        (B1) {r1 :  Val}
        (B2) {(truthand g1 r1) :  T
          & (truthor (append (list b) c a) r1) = nil}

       (@
         (Obviously {(truthand g1 r1) :  T & (truthor g2 r1) = nil})
         (We_see (Refutable g1 g2) (by  B1))
         (Hence CON))))))))


(LEMMA: PermL

  (Under_the_assumption

    (INDHYP0) {g1 g2 :  (Cartesian NonTrivFmL FmL)}

   (We_assume

    (INDHYP1:1) (a b c) matches (divide g1)
    (INDHYP1:2) (divide g1) exists
    (INDHYP1:3) {(EX p :  Prf)
                  (LET ((h1 h2) (con p)))
                     (h1 = (append (list b) c a) & h2 = g2)
                   +
                 (EX r :  Val)
                   ((truthand (append (list b) c a) r) :  T
                        & (truthor g2 r) = nil)}

   (We_prove CON (by_cases (A) or (B) of INDHYP1:3

     In_the_case (A)
                                                   ;TeX

      (Since A (we_may_assume there_exists p1 such_that

        (A1) {p1 :  Prf}
        (A2) {(LET ((h1 h2) (con p1)))
                 (h1 = (append (list b) c a) & h2 = g2)}

      (We_see CON (since (obviously (Provable g1 g2)
       regarding (list 'PermL (list g1 g2) p1) as p)))))
```

```
                                                          ;TeX
    In_the_case (B)

     (Since B (we_may_assume there_exists r1 such_that

       (B1) {r1 :  Val}
       (B2) {(truthand (append (list b) c a) r1) :  T
                  & (truthor g2 r1) = nil}

     (@
       (Obviously {(truthand g1 r1) :  T & (truthor g2 r1) = nil})
       (We_see (Refutable g1 g2) (by  B1))
       (Hence CON))))))))


(LEMMA:  Partial_correctness

  (Apply_simultaneous_induction_on

    {g1 g2 :  Right} {g1 g2 :  Left}
    {g1 g2 :  RuleR} {g1 g2 :  RuleL}
    {g1 g2 :  PermR} {g1 g2 :  PermL}

    to

    (the_proofs Right1 Right2)
    (the_proofs Left1 Left2)
    (the_proofs RuleR)
    (the_proofs RuleL)
    (the_proofs PermR)
    (the_proofs PermL)))

(LEMMA: Termination

   (Under_the_assumption (*) {g1 g2 :  Seq} (obviously {g1 g2 :  Right})))


(THEOREM: Completeness_theorem

 (Under_the_assumption (*) {g1 g2 :  Seq}

  (We_prove CON

  ;PROOF

  (@
    (We_see {g1 g2 :  Right} (by Termination))
    (By Partial_correctness
     (we_see `{g1 g2 :  Right -> _(Provable g1 g2) + _(Refutable g1 g2)}))
    (Hence CON)))))
```

In the abstract syntax, the completeness theorem is

```
Wang.
  {g1 g2 :  Seq ->
    (EX p :  Prf) (LET ((h1 h2) (con p))) (h1 = g1 & h2 = g2)
    + (EX r :  Val) ((truthand g1 r) :  T & (truthor g2 r) = nil)}

  with 27 hypotheses
```

The extracted program is

```
Function definitions are as follows:

(defrec <PermL-6>
        (g1 g2)
        nil
        (let!  (((@17:1 @17:2)
                 (let!  (((a b c) (divide g1)))
                       (<RuleL-4> (append (list b) c a) g2))))
           (case @17:1
              (list 1 (list 'PermL (list g1 g2) @17:2))
              (list 2 @17:2)))))

(defrec <PermR-5>
        (g1 g2)
        nil
        (let!  (((@13:1 @13:2)
                 (let!  (((a b c) (divide g2)))
                       (<RuleR-3> g1 (append (list b) c a)))))
           (case @13:1
              (list 1 (list 'PermR (list g1 g2) @13:2))
              (list 2 @13:2)))))

(defrec <RuleL-4>
        (g1 g2)
        nil
        (let!  (((@7:1 @7:2)
                 (let!  ((((a b) .  c) g1))
                       (list (<Right-0> c (cons a g2))
                             (<Right-0> (cons b c) g2)))))
           (let!  (((<disjE-4> <disjE-5>) @7:1))
              (case <disjE-4>
                 (let!  (((<disjE-2> <disjE-3>) @7:2))
                    (case <disjE-2>
                          (list 1
                             (list 'impE
                                (list g1 g2)
                                <disjE-5>
                                <disjE-3>))
                          (list 2 <disjE-3>)))
                 (list 2 <disjE-5>))))))

(defrec <RuleR-3>
        (g1 g2)
        nil
        (let!  (((@3:1 @3:2)
                (let!  ((((a b) .  c) g2))
                             (<Right-0> (cons a g1) (cons b c)))))
           (case @3:1
              (list 1 (list 'impI (list g1 g2) @3:2))
              (list 2 @3:2)))))

(defrec <Left-1>
        (g1 g2)
        nil
        (cond ((and g1 (existNonAtom g1)) (<PermL-6> g1 g2))
              (t
               (cond ((member nil g1)
                      (list 1 (list 'falsehood (list g1 g2))))
                     (t
                      (case (<List-2> g1 g2)
                            (list 1
                                  (list 'intersection
                                        (list g1 g2)))
                            (list 2
                                  (Lambda ((g1 g1))
                                          (lambda (a)
                                                  (cond
                                                   ((member a g1) t)))
                                          ))))))))

(defrec <Right-0>
        (g1 g2)
        nil
        (cond ((and g2 (existNonAtom g2)) (<PermR-5> g1 g2))
              (t (<Left-1> g1 g2))))

(defrec <List-2>
        (g1)
        (g2)
        (cond ((dtpr g1)
               (case (<List-2> (cdr g1))
                     1
                     (cond ((member (car g1) g2) 1) (t 2))))
              (t 2)))


The extracted realizer is

(Lambda ((g1 g1) (g2 g2)) (lambda nil (<Right-0> g1 g2)))
```

As was indicated above, the proof of the completeness theorem has 27 hypotheses. We will illustrate how to prove these in EKL by the following three examples. The examples displayed below have been taken from transcriptions of actual sessions, but some displays of EKL terms have been edited for typographical reasons.

(Example 1)

$$\nabla(h1\ h2) = con(list(\text{'}falsehood, list(g1, g2)).h1 = g1 \wedge h2 = g2.$$

The translated form of this formula is named `HYP` in EKL by declaring it as a propositional constant. Its conclusion is named by a propositional function `HYP_C`. The lines declaring these are labeled with `PRF`. We will show them with the command `(show prf)` as follows:

```
140.  (show prf)
;labels:  PRF HYP_C
137.  (DEFAX HYP_C
             |HYP_C(G1,G2) IFF
              CDDR(CON(LIST('FALSEHOOD,LIST(G1,G2)))) = NIL&
              (LAMBDA H1 H2.H1 = G1&H2 = G2)
              (CAR(CON(LIST('FALSEHOOD,LIST(G1,G2)))),
               CADR(CON(LIST('FALSEHOOD,LIST(G1,G2)))))|)

;labels:  PRF HYP
139.  (DEFAX HYP |HYP IFF HYP_C(G1,G2)|)
```

First, we open our target.

```
140.  (trw hyp (open hyp hyp_c))
;HYP IFF
;CDDR(CON(LIST('FALSEHOOD,LIST(G1,G2)))) = NIL&
;(LAMBDA H1 H2.H1 = G1&H2 = G2)
;(CAR(CON(LIST('FALSEHOOD,LIST(G1,G2)))),CADR(CON(LIST('FALSEHOOD,LIST(G1,G2)))))
```

We expand two expressions of the form `LIST(...)`. Since one is a subexpression of the other, we must open it twice.

```
141.  (rw * (open list list))
;HYP IFF
;CDDR(CON('FALSEHOOD.((G1.(G2.NIL)).NIL))) = NIL&
;(LAMBDA H1 H2.H1 = G1&H2 = G2)
;(CAR(CON('FALSEHOOD.((G1.(G2.NIL)).NIL))),
; CADR(CON('FALSEHOOD.((G1.(G2.NIL)).NIL))))
```

We expand `CON(...)` by `(use CON mode exact)`. When we take the option `(... mode exact)`, EKL apply the rewrite rule to each subexpression exactly once.

Otherwise, EKL applies it only if the result of the application is syntactically simpler than the original one. Because `CON` was introduced without using (`defax` ...), we cannot use (`open con`) (The reason why we do not use (`defax` ...) is that it may refer to itself in its definition). Since `CON` is a strict function, we must show that its arguments exist. To this end, we add `tsf2` to the rewriter. `tsf2` is an axiom maintaining that any functions of sort `tsf2` are total and strict. Since `CONS`, which is denoted as infixed "`.`", has the sort `tsf2`, the arguments of `CON` exist.

```
142.  (rw * (use tsf2) (use con mode exact))
;HYP IFF
;CDDR(CADR(‘FALSEHOOD.((G1.(G2.NIL)).NIL))) = NIL&
;(LAMBDA H1 H2.H1 = G1&H2 = G2)
;(CAR(CADR(‘FALSEHOOD.((G1.(G2.NIL)).NIL))),CADR(CADR(‘FALSEHOOD.((G1.(G2.NIL)).NIL))))
```

We expand `CADR` and `CDDR`. We open `CADR` twice for the same reason as mentioned above.

```
143.  (rw * (open cadr cadr cddr))
;HYP IFF
;CDR(CDR(CAR(CDR(‘FALSEHOOD.((G1.(G2.NIL)).NIL))))) = NIL&
;(LAMBDA H1 H2.H1 = G1&H2 = G2)
;(CAR(CAR(CDR(‘FALSEHOOD.((G1.(G2.NIL)).NIL)))),
; CAR(CDR(CAR(CDR(‘FALSEHOOD.((G1.(G2.NIL)).NIL))))))
```

Finally, we reduce `CAR(CONS(`...`))` and `CDR(CONS(`...`))` by the axioms `car_cons` and `cdr_cons`, which maintain that `CAR` and `CDR` are destructors of pairs constructed by `CONS`. The necessary condition of this reduction such that the two arguments of `CONS` exist is guaranteed by `tsf2`.

```
144.  (rw * (use tsf2 car_cons cdr_cons))
;HYP

145.  (qed)
```

We can prove the conjecture in one line as follows.

```
140.  (trw hyp (open hyp hyp_c list list cadr cadr cddr) (use tsf2 car_cons cdr_cons)
               (use con mode exact))
;HYP

141.  (qed)
```

(Example 2)

$$member(nil, g1) : T \wedge [g1, g2] : Cartesian(FmL, TrivFmL)$$
$$\supset list(\text{'}falsehood, list(g1, g2)) : Prf$$

Each assumption is automatically assumed and labeled with `HYP_`n, where n is its serial number. It is labeled with `HYP_a`, too. So `HYP_a` is a label for all assumptions.

```
276. (show prf)
;labels: PRF HYP_A HYP_1
270. (ASSUME |MEMBER(NIL,G1) :: _T|)
;deps: (HYP_1)

;labels: PRF HYP_A HYP_2
271. (ASSUME |(G1,G2) :: _CARTESIAN(_FM_L,_TRIV_FM_L)|)
;deps: (HYP_2)

;labels: PRF HYP_C
273. (DEFAX HYP_C |HYP_C(G1,G2) IFF LIST('FALSEHOOD,LIST(G1,G2)) :: _PRF|)

;labels: PRF HYP
275. (DEFAX HYP |HYP IFF (MEMBER(NIL,G1) :: _T&
                        (G1,G2) :: _CARTESIAN(_FM_L,_TRIV_FM_L) IMP HYP_C(G1,G2))|)

276. (trw |hyp_c(g1,g2)| (open hyp_c))
;HYP_C(G1,G2) IFF LIST('FALSEHOOD,LIST(G1,G2)) :: _PRF
```

First, we expand `_PRF` using axiom `_prf_fix`. `_prf_fix` is an axiom maintaining that class `_PRF` is a fixed point of its CIG inductive definition.

```
277. (rw * (use tsfn) (use _prf_fix mode exact))
;HYP_C(G1,G2) IFF
;LIST('FALSEHOOD,LIST(G1,G2)) :: _DP&
;CONDFML((EQUAL(CAR(LIST('FALSEHOOD,LIST(G1,G2))),'PERMUTATION), ...)
;         (EQUAL(CAR(LIST('FALSEHOOD,LIST(G1,G2))),'INTERSECT), ...)
;         (EQUAL(CAR(LIST('FALSEHOOD,LIST(G1,G2))),'FALSEHOOD),
;          CDDADR(LIST('FALSEHOOD,LIST(G1,G2))) = NIL&
;          (LAMBDA A B.A :: _FM_L&B :: _FM_L&MEMBER(NIL,A) :: _T)
;          (CAADR(LIST('FALSEHOOD,LIST(G1,G2))),
;           CADADR(LIST('FALSEHOOD,LIST(G1,G2))))),
;         (EQUAL(CAR(LIST('FALSEHOOD,LIST(G1,G2))),'IMP_I), ...)
;         EQUAL(CAR(LIST('FALSEHOOD,LIST(G1,G2))),'IMP_E), ...)
```

We simplify (`COND` ...) by axiom `condfml` after opening `LIST` and `EQUAL`. At the same time, we expand `_DP`.

```
278.  (rw * (use tsfn tsf2 car_cons condfml) (open equal list) (use _dp_fix_mode exact))
;HYP_C(G1,G2) IFF
;NOT DTPR(‘FALSEHOOD.((G1.(G2.NIL)).NIL)) = NIL&CDDADR(‘FALSEHOOD.(LIST(G1,G2).NIL)) = NIL&
;(LAMBDA A B.A ::  _FM_L&B ::  _FM_L&MEMBER(NIL,A) ::  _T)
;(CAADR(‘FALSEHOOD.(LIST(G1,G2).NIL)),CADADR(‘FALSEHOOD.(LIST(G1,G2).NIL)))
```

Note that `DTPR` is defined using `ATOM`. We open it and execute reduction of `CAR` and `CDR`.

```
279.  (rw * (use tsf2 atom_cons car_cons cdr_cons condexp)
              (open dtpr list cddadr caadr cadadr))
;HYP_C(G1,G2) IFF G1 ::  _FM_L&G2 ::  _FM_L&MEMBER(NIL,G1) ::  _T
```

The third conjunct is implied by the line `HYP_1`, so we can delete it.

```
280.  (rw * (use hyp_a))
;HYP_C(G1,G2) IFF G1 ::  _FM_L&G2 ::  _FM_L
;deps:  (HYP_A)

284.  (label target)
;Labeled.
```

Labeling it, we now rewrite the line `HYP_2`. Then we apply the result to the target.

```
284.  (rw hyp_2 (use _cartesian_fix _triv_fm_l_fix mode exact))
;G1 ::  _FM_L&G2 ::  _FM_L&NOT G2 ::  _NON_TRIV_FM_L
;deps:  (HYP_2)

285.  (rw target (use *))
;HYP_C(G1,G2)
;deps:  (HYP_A)
```

We now get the conjecture `HYP`. The command (`ci` ...) introduces implication by discharging assumptions.

```
286.  (ci hyp_a *)
;MEMBER(NIL,G1) ::  _T&(G1,G2) ::  _CARTESIAN(_FM_L,_TRIV_FM_L) IMP HYP_C(G1,G2)

287.  (trw hyp (open hyp) (use *))
;HYP
```

```
288.  (qed)
```

(Example 3)

$$member(nil, g1) = nil \supset \Lambda(g1 = g1)(\lambda(a)(cond(member(a, g1), t))) : Val$$

```
153.  (show prf)
;labels:  PRF HYP_A HYP_1
148.  (ASSUME |MEMBER(NIL,G1) = NIL|)
;deps:  (HYP_1)

;labels:  PRF HYP_C
150.  (DEFAX HYP_C |HYP_C(G1) IFF CLOSURE1(G1) ::  _VAL|)

;labels:  PRF HYP
152.  (DEFAX HYP |HYP IFF (MEMBER(NIL,G1) = NIL IMP HYP_C(G1))|)
```

The axiom concerning CLOSURE1 is as follows :

```
153.  (show closure1)
;labels:  CLOSURE1
144.  (AXIOM |ALL G1 ARG_CLOSURE1.
                E(CLOSURE1(G1))&
                APPLY(CLOSURE1(G1),LIST(ARG_CLOSURE1)) =
                (LAMBDA A.COND(MEMBER(A,G1),T))(ARG_CLOSURE1)|)
```

We open HYP and _VAL first.

```
153.  (trw |hyp_c(g1)| (open hyp_c) (use closure1) (use _val_fix mode exact))
;HYP_C(G1) IFF
;CLOSURE1(G1) ::  _FUNC(_ATM,_BOOL)&FUNCALL(CLOSURE1(G1),NIL) = NIL

154.  (rw * (use closure1 _func_fix mode exact) (open funcall))
;HYP_C(G1) IFF
;(ALL X.X ::  _ATM IMP COND(MEMBER(X,G1),T) ::  _BOOL)&COND(MEMBER(NIL,G1),T) = NIL

155.  (rw * (use hyp_a condexp))
;HYP_C(G1) IFF (ALL X.X ::  _ATM IMP COND(MEMBER(X,G1),T) ::  _BOOL)
;deps:  (HYP_1)

156.  (label target)
```

```
;Labeled.
```

We prove it by cases. The first case is

```
156.  (assume |not member(x,g1) = nil|)
;deps:  (156)

157.  (trw |cond(member(x,g1),t) ::  _bool| (use * condexp tsf2)
                                             (use _bool_fix mode exact))
;COND(MEMBER(X,G1),T) ::  _BOOL IFF CONDFML((T,TRUE),T,TRUE)
;deps:  (156)

158.  (rw * (use condfml))
;COND(MEMBER(X,G1),T) ::  _BOOL
;deps:  (156)

159.  (label case1)
;Labeled.
```

The second case is

```
159.  (assume |member(x,g1) = nil|)
;deps:  (159)

160.  (trw |cond(member(x,g1),t) ::  _bool| (use * condexp condfml)
                                            (use _bool_fix mode exact))
;COND(MEMBER(X,G1),T) ::  _BOOL
;deps:  (159)

161.  (label case2)
;Labeled.
```

We now merge the two subproofs.

```
161.  (derive |not member(x,g1) = nil or member(x,g1) = nil|)

162.  (cases * case1 case2)
;COND(MEMBER(X,G1),T) ::  _BOOL

163.  (rw target (use *))
;HYP_C(G1)
;deps:  (HYP_1)

164.  (ci hyp_a *)
```

```
;MEMBER(NIL,G1) = NIL IMP HYP_C(G1)
```

We can now obtain `HYP`.

```
165.  (trw hyp (open hyp) (use *))
;HYP
```

The following shows how the TEX translator translates the portion of the proof that begins and ends with the lines flagged by TEX.

Since (A), we may assume there exists $p1$ such that

(A1) $$p1 : Prf$$

(A2) $$\nabla(h1\ h2) = con(p1).\ h1 = append(list(b), c, a) \wedge h2 = g2$$

We see (CON), since obviously $Provable(g1, g2)$ regarding $list('PermL, list(g1, g2), p1)$ as p.

The TEX translator does not respect overfill of lines, so one may have overfull lines.

C Optimizers

The following are definitions of optimizers and **subst** used in section 3.2.

$$\begin{gathered}
\mathit{Optimize}_{case}(\mathit{case}(i, e_1, \ldots, e_n)) \longmapsto e_i, \\
\mathit{Optimize}_{case}(\mathit{case}(\mathit{case}(e, 1, \ldots, n)), e_1, \ldots, e_n) \longmapsto e, \\
\mathit{Optimize}_{\beta}(\mathit{app}(\Lambda(\lambda(a_1, \ldots, a_n).e), e_1)) \longmapsto \textbf{subst}\ [a_1, \ldots, a_n] \leftarrow e_1\ \textbf{in}\ e, \\
\mathit{Optimize}_{\beta}(\mathit{app}{*}(\Lambda(\lambda(a_1, \ldots, a_n).e), e_1, \ldots, e_n)) \\
\longmapsto \textbf{subst}\ a_1 \leftarrow e_1, \ldots, a_n \leftarrow e_n\ \textbf{in}\ e, \\
\mathit{Optimize}_{\eta}(\Lambda(\lambda(a_1, \ldots, a_n).\mathit{app}(e, \mathit{list}(a_1, \ldots, a_n)))) \longmapsto e \\
\mathit{Optimize}_{\eta}(\Lambda(\lambda(a_1, \ldots, a_n).\mathit{app}{*}(e, a_1, \ldots, a_n))) \longmapsto e
\end{gathered}$$

The **subst** construct does some substitution and/or builds a *let*-form: Let each of $\alpha_1, \ldots, \alpha_n$ be a "binding" $p = e$ or a "substitution" $[a_1, \ldots, a_n] \leftarrow e$ or $(a_1 \ldots a_n\ .\ b) \leftarrow e$. Then we define **subst** $\alpha_1, \ldots, \alpha_n$ **in** e. Let $p \leftarrow e_1$ be one of the assignments of $\alpha_1, \ldots, \alpha_n$. Let $a_1, \ldots, a_n$ be $FV(p)$. Then we define a term e^* and a sequence of bindings, say B, by the the following cases.

(1) None of the $a_1, \ldots, a_n$ appear freely in e. Then B is empty and e' is e itself. Note that n may be 0, e.g., p is the empty pattern (). That case is recognized as a particular case of the present case.

(2) e has a free occurrence of a variable of $a_1, \ldots, a_n$. Let *new* be a new total individual variable. If p has the form $[a_1, \ldots, a_n]$, then let e' be the expression obtained from e by replacing the free occurrences of $[a_1, \ldots, a_n]$ in e by *new*. Otherwise e' is e itself. (Note that each of the a_i must not be bounded in a *free* occurrence of $[a_1, \ldots, a_n]$.) Let e'' be $e'[e_1/\mathit{new}]$, if *new* occurs at most once in e'. Otherwise e'' is e' itself. Let B'' be $\emptyset$, if *new* occurs at most once in e'. Otherwise B'' is $\mathit{new} = e_1$.

Let e_1 be the form $\mathit{pair}(d_1, \mathit{pair}(d_2, \cdots \mathit{pair}(d_u, e_2) \cdots))$, possibly $u = 0$, such that e_2 is not the form $\mathit{pair}(*, *)$. We consider $\mathit{list}(d_1, \ldots, d_n)$ as

$$\mathit{pair}(d_1, \mathit{pair}(d_2, \cdots \mathit{pair}(d_u, \mathit{nil}) \cdots)).$$

Let $\mathit{expand}'(p)$ be the expansion of p except that *list* is rewritten by successive applications of *pair*. $match(expand'(p), e)$ is a set of bindings defined by

$$match(expand'(p), e) = \begin{cases} match(d_1, e_1) \cup match(d_2, e_2), & \text{if } e \equiv pair(e_1, e_2) \\ & \quad \text{and} \\ & \quad expand'(p) \\ & \qquad \equiv pair(d_1, d_2) \\ \emptyset, & \text{if } p = [\,] \\ \{p = e\} & \text{otherwise} \end{cases}$$

Then we define e^* by cases.

(2a) There is a free occurrence of $a_1, \ldots, a_n$ in e'', but *new* does not occur. Let $p_1 = d_1, \ldots, p_u = d_u$ be the bindings among $match(expand'(p), e_1)$ such that p_i is a *single variable* and appears freely only once in e_1 or d_i is a variable or a constant, and let $p'_1 = d'_1, \ldots, p'_m = d'_m$ be the union of the other bindings and B''. Then e^* is

$$let\ p'_1 = d'_1, \ldots, p'_m = d'_m\ in\ e_1[d_1/p_1, \ldots, d_u/p_u].$$

(2b) There is a free occurrence of $a_1, \ldots, a_n$ in e'' and *new* occurs, too. Then e^* is

$$let\ new = e_1\ in\ let\ p = new\ in\ e_2.$$

Further optimization is possible in the case of (2b), but we do not carry it out since it is rather messy.

The power of our optimization method is rather limited, since optimizers are not defined recursively. When $Optimize_\beta$ is applied to a β-redex, then it may create a new β-redex. $Optimize_\beta$ does not reduce such a new β-redex. Examples of such β-redexes appeared in an extracted program in 4.3.2.6. So improvement of optimizers should be done in a future study. The correctness of optimizations of arbitrary program is not quite easy in the type free approach. For example, the optimization of $\Lambda(\lambda(x)(app{*}(\Lambda(\lambda(y)(e_1)), e_2)))$ to $\Lambda(\lambda(x)(e_1[e_2/y]))$ is not always correct, since their values may be different in our semantics of Λ-form. Generally speaking, optimization in Λ-form is not correct. But, if

$$\Lambda(\lambda(x)(app{*}(\Lambda(\lambda(y)(e_1)), e_2)))$$

is created as a realizer of $\forall x : D.A$, then the optimized form $\Lambda(\lambda(x)(e_1[e_2/y]))$ is also a realizer of $\forall x : D.A$, since $app{*}(\Lambda(\lambda(y)(e_1)), e_2)$ equals $e_1[e_2/y]$. Namely, we have to consider how an expression is used to do optimization. In type theories, expressions are typed and their types describe how they are used. So optimizations are easier in type theories. The difficulty of optimization is a drawback in the type free approach of **PX**. Further research is necessary to solve this problem.

References

Alagić, S. and Arbib, M.A.
1978 *The Design of Well-Structured and Correct Programs*, Springer-Verlag.

Allen, S.
1987 A Non-Type-Theoretic Definition of Martin-Löf's Types, *Proceedings of the 2nd Symposium on Logic in Computer Science*, IEEE, pp. 215-221

Apt, K.R.
1981 Ten Years of Hoare's Logic: A Survey-Part I, *ACM Transactions of Programming Languages and Systems*, vol. 3, pp. 431-483.

Backus, J.
1978 Can programming be liberated from the von Neumann Style?, *Communications of ACM*, vol. 21, pp. 613-641.

Barwise, J.
1977 *Handbook of Mathematical Logic*, North-Holland, Amsterdam.

Bates, J.L., and Constable, R.L.
1985 Proofs as programs, *ACM Transactions of Programming Languages and Systems*, vol. 7, no. 1, pp. 113-136.

Beeson, M.J.
1978 A type-free Gödel interpretation, *The Journal of Symbolic Logic*, vol. 43, no. 2, pp. 213-246.

1981 Formalizing constructive mathematics: why and how?, *Lecture Note in Mathematics*, Springer-Verlag, vol. 873, pp. 146-190.

1985 *The Foundations of Constructive Mathematics*, Springer-Verlag.

1986 Proving Programs and Programming Proofs, in *Logic, Methodology, and Philosophy of Sciences VII*, Barcan Marcus, R., Dorn, G.J.W., and Weingartner, P., eds., North-Holland, Amsterdam, pp. 51-82.

Bishop, E.
1967 *Foundations of Constructive Analysis*, McGraw-Hill.

1970 Mathematics as a numerical language, in *Intuitionism and Proof theory: Proceedings of the Summer Conference at Buffalo, New York, 1968*, Kino, A., Myhill, J., and Vesley, R.E., eds., North-Holland, pp. 53-71.

Bishop, E. and Bridges, D.
1985 *Constructive Analysis*, Springer-Verlag.

Boyer, R.S., and Moore, J.S.
1979 *A computational Logic*, Academic Press.

Burstall, R., and Lampson, B.
1984 A kernel language for abstract data types and modules, *Lecture Note in Computer Science*, Springer-Verlag, vol. 173, pp. 1-50.

Cardelli, L.
1986 *A polymorphic λ-calculus with Type:Type*, preprint, Systems Research Center, Digital Equipment Corporation.

Cardelli, L., and Wegner, P.
1985 On understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Survey*, vol. 17, no. 4, pp. 471-522.

Chang, C.C., and Keisler, H.J.
1973 *Model theory*, second edition, North-Holland.

Constable, R.L., and Mendler, N.P.
1985 Recursive Definitions in Type Theory, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 193, pp. 61-78.

Constable, R.L., et al.
1986 *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall.

Coppo, M., and Dezani-Ciancaglini, M.
1979 A new type assignment for λ-terms, *Archive für mathematisch Logik und Grundragenforschung*, vol. 19, pp. 139-156.

Coquand, T.
1986 An Analysis of Girard's Paradox, Rapports de Recherche, INRIA, no. 531.

Coquand, T., and Huet, G.
1985 A Selected Bibliography on Constructive Mathematics, Intuitionistic Type Theory and Higher Order Deduction, *J. Symbolic Computation*, vol. 1, pp. 323-328.

1986 The Calculus of Constructions, Rapports de Recherche, INRIA, no. 530, *Information and Computation*, to appear.

Cousineau, G, Curien, P.L., and Mauny, M.
1985 The Categorical Abstract Machine, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 201, pp. 50-64.

Feferman, S.
1975 A language and axioms for explicit mathematics, *Lecture Notes in Mathematics*, Springer-Verlag, vol. 450, pp. 87-139.

1979 Constructive theories of functions and classes, in *Logic Colloquium '78*, Boffa, M., and van Dalen, D., and McAloon, K., eds., North-Holland, pp. 159-224.

1984 Between constructive mathematics and classical mathematics, *Lecture Notes in Mathematics*, Springer-Verlag, vol. 1104, pp. 143-161.

Foderaro, J.K., Sklower, K.L., and Layer, K.
1984 *The Franz Lisp Manual.*

Fourman, M.P.
1977 The logic of topoi, in *Handbook of Mathematical Logic*, Barwise, J., ed., North-Holland, pp. 1053-1090.

Friedman, H.M.

1971 Axiomatic recursive function theory, in *Logic Colloquium 69*, Gandy, R.O., and Yates, C.M.E., eds., North-Holland, pp. 113-137.

Furukawa, K. and Yokoi, T.

1984 Basic Software System, in *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, pp. 37-57.

Girard, J.Y.

1972 *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Ph.D. thesis, University of Paris, VII.

Goad, C.

1980 *Computational uses of the manipulation of formal proofs*, Ph.D. thesis, Stanford University.

Gordon, M.J., Milner, R., and Wadsworth, C.P.

1979 *Edinburgh LCF*, Lecture Notes in Computer Science, Springer-Verlag, vol. 78.

Goto, S.

1979 Program synthesis from natural deduction proofs, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, vol. 1, pp. 339-341.

1979a Program synthesis through Gödel's interpretation, *Lecture Notes in Computer Science*, vol. 75, pp. 302-325.

Hagiya, M., and Sakurai, T.

1984 Foundations of Logic Programming Based on Inductive Definition, *New Generation Computing*, vol.2, pp. 59-77.

Hayashi, S.

1982 A note on bar induction rule, in *The L.E.J. Brouwer Centenary Symposium*, Troelstra, A.S., and van Dalen, D. eds., North-Holland, pp. 149-163.

1983 Extracting Lisp Programs from Constructive Proofs: A Formal Theory of Constructive Mathematics Based on Lisp, *Publications of the Research Institute for Mathematical Sciences*, vol. 19, pp. 169-191.

1986 **PX**: a system extracting programs from proofs, in *Formal Description of Programming Concepts-III*, Wirsing, M., ed., North-Holland.

Hyland, J.M.E.

1982 The Effective Topos, in *The L.E.J. Brouwer Centenary Symposium*, Troelstra, A.S., and van Dalen, D., eds., North-Holland, pp. 165-216.

Johnstone, P.T.

1977 *Topos theory*, Academic Press.

Ketonen, J., and Weening, J.S.

1983 *The Language of an Interactive Proof Checker*, Stanford University, report no. STAN-CS-83-992.

1984 *EKL-An Interactive Proof Checker User's Reference Manual*, Stanford University.

Kleene, S.C.
1952 *Introduction to Metamathematics*, Van Nostrand.

Lambek, J.L., and Scott, P.J.
1986 *Introduction to higher order categorical logic*, Cambridge University Press.

MacQueen, D.
1986 Using Dependent Types to Express Modular Structure, in *13th Annual Symp. on Principles of Programming Languages*, pp. 277-286.

MacQueen, D., and Sethi, R.
1982 A Semantic Model of Types for Applicative Languages, in *ACM Symp. on Lisp and Functional Programming, Pittsburgh*, pp. 243-252.

Manna, Z.
1974 *Mathematical Theory of Computation*, McGraw-Hill.

Manna, Z., and Waldinger, R.
1971 Towards automatic program synthesis, *Communications of ACM*, vol. 14, pp. 151-165.

Martin-Löf, P.
1982 Constructive mathematics and computer programming, in *Logic, Methodology, and Philosophy of Science VI*, Cohen, L.J., et al., eds., North-Holland, pp. 153-179.
1983 *On the meanings of the logical constants and the justifications of the logical laws*, lectures given at Siena, April 1983, included in Proceedings of the Third Japanese-Swedish Workshop, Institute of New Generation Computing Technology, 1985.

McCarthy, J., et al.
1965 *Lisp 1.5 Programmer's Manual*, MIT Press.

McCarty, D.C.
1984 *Realizability and Recursive Mathematics*, Ph. D. thesis, Oxford.

Meyer, A.R., and Reinhold, M.B.
1986 'Type' is not a type, in *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery.

Mitchell, J.C.
1986 A type-inference approach to reduction properties and semantics of polymorphic expressions, in *1986 ACM Symposium on Lisp and Functional Programming*, pp. 308-319.

Mitchell, J.C., and Plotkin, G.
1985 Abstract data types have existential type, in *12th Annual Symp. on Principles of Programming Languages*, pp. 37-51.

Moggi, E.
1986 Categories of partial morphisms and the partial lambda calculus, in *Lecture Notes in Computer Science*, Springer-Verlag, vol. 240, pp. 242-251.

1988 The Partial Lambda-Calculus, Ph.D. thesis, University of Edinburgh.

Nakajima, R., and Yuasa, T.

1983 *The IOTA Programming System*, Lecture Notes in Computer Science, vol. 160.

Nepeĭvoda, N.N.

1978 A relation between the natural deduction rules and operators of higher level algorithmic languages, *Soviet Mathematics Doklady*, vol.19, no. 2, pp. 360-363

1982 Logical Approach to Programming, in *Logic, Methodology, and Philosophy of Science VI*, Cohen, L.J., et al., eds., North-Holland, pp. 109-122.

Nordström, B., and Petersson, K.

1983 Programming in constructive set theory: some examples, in *Proceedings of 1981 Conference on Functional Programming Language and Computer Architecture*, pp. 141-153.

Plotkin, G.

1985 Lectures given at ASL Stanford meeting, July.

Reynolds, J.C.

1985 Three approaches to type structure, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 185, pp. 97-138.

Sato, M.

1979 Towards a mathematical theory of program synthesis, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, vol. 2, pp. 757-762.

1985 *Typed Logical Calculus*, Department of Information Science, Faculty of Science, University of Tokyo, Technical Report 85-13.

Scott, D.S.

1979 Identity and existence in intuitionistic logic, *Lecture Notes in Mathematics*, Springer-Verlag, vol. 753, pp. 660-696.

Shankar, N.

1985 Towards Mechanical Metamathematics, *Journal of Automated Reasoning*, vol. 1, pp. 407-434.

Shockley, J.E.

1967 Introduction to Number Theory, Holt, Rinehart and Winston.

Smith, J.

1984 An interpretation of Martin-Löf's type theory in a type-free theory of propositions, *The Journal of symbolic logic*, vol. 49, no. 3, pp. 730-753.

Steele, G.

1984 *Common LISP: The language*, Digital Press.

Takasu, S., and Kawabata, S.

1981 A Logical Basis for programming Methodology, *Theoretical Computer Science*, vol. 16, pp. 43-60.

Takasu, S., and Nakahara, T.
1983 Programming with mathematical thinking, in *IFIP 83*, North-Holland, pp. 419-424.

Tatsuta, M.
1987 *Program Synthesis Using Realizability*, Master's thesis, University of Tokyo.

Troelstra, A.S.
1973 *Metamathematical investigations of intuitionistic arithmetic and analysis*, Lecture Notes in Mathematics, Springer-Verlag, vol. 344

Index