LAMBDA ENCODINGS IN TYPE THEORY

by

Peng Fu

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

August 2014

Thesis Supervisor: Associate Professor Aaron Stump

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Peng Fu

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the August 2014 graduation.

Thesis Committee: ______________________________
Aaron Stump, Thesis Supervisor

____
Cesare Tinelli

____
Kasturi R. Varadarajan

____
Ted Herman

____
Douglas W. Jones

To my mother Chen Xingzhen.

ACKNOWLEDGEMENTS

I would like to thank first of all my thesis advisor, Prof. Aaron Stump. Five years ago when I applied for graduate school, I asked him for recommendation on books in logic. He suggested Girard's *proofs and types* to me. It is not an ordinary book in logic and it introduces me to the Girard's works on System **F**, and the wonderful world of type theory and lambda encodings. I am also very grateful for his advice and support during my graduate study. The vivid discussions with him on research will be at a special place in my memory. I would also like to thank Prof. Cesare Tinelli. Even we have not have a chance yet to collaborate on research, I learned a lot of interesting things in automated reasoning from him in the CLC seminar, some of which even found their connections in this dissertation. Next, I would like to thank my dissertation committee, Prof. Kasturi Varadarajan, Prof. Ted Herman and Prof. Douglas Jones, for their time, patience and many interesting discussions on my research and thesis topic. I would also want to take this opportunity to thank Prof. Gregory Landini from the department of philosophy, I learn Frege and Russell' works and approach to foundations of mathematics from him. I also learned how to derive strong induction from weak induction from his class on mathematical logic. It would become obvious to see the philosophy origins of the last two Chapters of this dissertation. I spent four semesters in mathematics department, I am very proud of myself on being one of the students in Prof. Frauke Bleher's algebra classes. I hope that one day I can apply the knowlege I learned from algebra in my research. I am glad that I have the opportunity to learn Haskell programming and discussed some of language implementation issues with Garrin Kimmell. I am very fortunate to have the opportunity to meet and discuss research idea with Prof. Tim Sheard, KiYung Ahn, Nathan Collins from Portland State University; Prof. Stephanie Weirich, Vilhelm Sjöberg, Chris Casinghino from University of Pennsylvania. I am very grateful to have the opportunity to interact with my fellow labmates at lab 317, I would like to mention Harley Eades, Tianyi Liang, Duckki Oe, Ryan McCleeary and Andrew Reynolds. On a more personal side, I am a long term inhabitant of the main library in University of Iowa, part of the research in this dissertation is done in the space generously provided by the library, not to mention about the 66 books that I borrowed from the library. I also want to thank my friend Wang Tianfeng in China, who encouraged me to apply to the graduate school in United States. Finally and most importantly, I would like to thank my mother Chen Xingzhen, for her unconditional love and supports through out all these years. And my uncle Chen Xuxiang, who always encourages me to learn and explore. My graduate study would not be possible without their help.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Lambda encodings (such as Church encoding, Scott encoding and Parigot encoding) are methods to represent data in lambda calculus. Curry-Howard correspondence relates the formulas and proofs in intuitionistic logics to the types and programs in typed functional programming languages. Roughly speaking, Type theory (Intuitionistic Type Theory) formulates the intuitionistic logic in the style of typed functional programming language. This dissertation investigates the mechanisms to support lambda encodings in type theory. Type theory, for example, Calculus of Constructions (**CC**) does not directly support inductive data because the induction principle for the inductive data is proven to be not derivable. Thus inductive data together with inductive principle are added as primitive to **CC**, leading to several nontrivial extensions, e.g. Calculus of Inductive Constructions. In this dissertation, we explore alternatives to incorporate inductive data in type theory. We propose to consider adding an abstraction construct to the intuitionistic type to support lambda-encoded data, while still be able to derive the corresponding induction principle. The main benefit of this approach is that we obtain relatively simple systems, which are easier to analyze and implement.

1.1 Motivation

Inductively defined data type (inductive data), together with the *pattern matching* mechanism, are commonly used in theorem proving and functional programming.

Most typed functional programming languages and theorem provers (Haskell, OCaml, Agda [8], Coq [49], TRELLYS [30], [10]) support them as primitives. Usually, the concepts of inductive data and program are separated, one can only perform pattern matching on inductive data. In lambda calculus however, there are no distinctions between program and data. For example, for Church numeral 2, it can be used as a higher order function that takes in a function f and a data b as arguments, then applying f to b twice.

From the programming language design perspective, inductive datatype and pattern matching increase the complexity of desgin, analysis and implementation of the language. For example, the pattern matching *case*-expression is considered the most complicated part of the Haskell core language [1]. Despite this complication, there are two main reasons that the language designers choose primitive data type over lambda encoding.

1. Defining function by recursion seems more natural compare to defining function by iteration. For example, defining subdata accessor with pattern matching is almost trivial while it is a challenging programming task for Church encoding scheme [12].

2. Primitive data type and pattern matching fit well with Hindley-Milner polymorphic type inference ([28], [35]), which is a key component for most static typed functional languages. With Scott encoding and Parigot encoding scheme,

[1]`http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType`

it is not clear how to directly achieve decidable type inference.

We counter the first reason with Scott encoding scheme. It is well know that primitive data type and pattern matching can be reduced to Scott encoded data and recursive definitions in a direct way ([37], [16]) and subdata accessor can be defined easily with Scott encoding using recursion. For the second reason, we can use a surface language for type inference while use lambda calculus with recursive definitions as the untyped core language. Since type inference/checking never interfere with the actual execution of the program, it only affects how the program is written, once a program is accepted by the type checker, we can translate it to lambda calculus and execute it. So we think that the primitive data and pattern matching in a functional language can be reasonably reduced to lambda calculus with Scott encoding, which simplifies the execution model for the language. We implement these ideas in Gottlob (see Chapter 7), which empirically shows that these ideas are reasonable.

If one wants to design an interactive theorem prover based on intuitionistic type theory à la Martin-Löf [33], then it is desirable to interpret the inhabitant of the type $D \to D$ as a total function on inductive datatype D. This is hard to achieve with Scott encoding, since each Scott-encoded data contains a piece of its subdata, one would need recursive type definition to type these data and operations. It is well known every type is inhabited once we admit unrestrictive recursive type definition[2]. Church encoding may be more suitable for the intuitionistic typing, it is already typable in System **F**. Besides the efficiency issue we mentioned before, there are three

[2]Certain restrictions are possible to retain totality, see [40], [43] and [34].

problems that prevent Church encoding from being adopted in interactive theorem provers based on intuitionistic types.

1. One can not construct a proof of $0 \neq 1$ with Church encoding [51].

2. Induction principle is not derivable in extensions of System **F** such as Calculus of Construcitons (**CC**) [20].

3. Computing type from data is not possible with Church encoding.

For the first problem, we propose to change the notion of contradiction, and we show how to prove $0 \neq 1$ with this new notion of contradiction. For the second problem, we propose a new type construct called *self types* to derive induction principle. We will cover these two topics in depth in Chapter 5. For the third problem, we think it is a fundamental problem for Church encoding in intuitionistic type theory due to the Girard's paradox [24]: in order to compute type from Church numerals, we would need to impredicative polymorhism at kind level, which is known to be inconsistent. One common practice to avoid this kind of problem is adopting infinite predicative hierarchy, which is beyond the scope of this dissertation.

The dissertation first describes fundamental concepts (Chapter 2) such as lambda encodings, abstract rewrite system and confluence. Then we discuss the confluence problem for lambda-mu calculus (Chapter 3). In Chapter 4, we show a limited way to construct expressive type theory based on the notion of *internalization.* System **S** is presented in Chapter 5, we introduce *self type* construct and use it to derive induction principle. Metatheorems such as consistency and type preservation are proved. In Chapter 6, System $\mathfrak{G}$ is presented, which is based on interpreting the iota-binder as set abstraction. Unlike System **S**, $\mathfrak{G}$ does not require recursive definition to describe induction principle, this simplifies the meta-theoretic property of $\mathfrak{G}$. We show that $\mathfrak{G}$ is consistent and we demonstrate some applications and some special properties of $\mathfrak{G}$. Finally, Chapter 7 discusses the design and implemented features of Gottlob. The logic of Gottlob is an extension of $\mathfrak{G}$. Future improvements of Gottlob are also discussed.

CHAPTER 2

PRELIMINARIES

In this Chapter, we first introduce abstract reduction system. Then, we review three lambda encoding schemes, namely, Church encoding, Scott encoding and Parigot encoding. Finally, we discuss the confluence property, which is a key property for abstract reduction system, including lambda calculus.

2.1 Abstract Reduction System

Definition 1. *An abstract reduction system $\mathcal{R}$ is a tuple $(\mathcal{A}, \{\to_i\}_{i\in\mathcal{I}})$, where $\mathcal{A}$ is a set and $\to_i$ is a binary relation(called reduction) on $\mathcal{A}$ indexed by a finite nonempty set $\mathcal{I}$.*

In an abstract reduction system $\mathcal{R}$, we write $a \to_i b$ if $a, b \in \mathcal{A}$ satisfy the relation $\to_i$. For convenient, $\to_i$ denotes a subset of $\mathcal{A} \times \mathcal{A}$ such that $(a, b) \in \to_i$ if $a \to_i b$.

Definition 2. *Given abstract reduction system $(\mathcal{A}, \{\to_i\}_{i\in\mathcal{I}})$, the reflexive transitive closure of $\to_i$ is written as $\twoheadrightarrow_i$ or $\overset{*}{\to}_i$, is defined by:*

- *$m \twoheadrightarrow_i m$.*
- *$m \twoheadrightarrow_i n$ if $m \to_i n$.*
- *$m \twoheadrightarrow_i l$ if $m \twoheadrightarrow_i n, n \twoheadrightarrow_i l$.*

Definition 3. *Given abstract reduction system $(\mathcal{A}, \{\to_i\}_{i\in\mathcal{I}})$, the convertibility relation $=_i$ is defined as the equivalence relation generated by $\to_i$:*

- $m =_i n$ *if* $m \twoheadrightarrow_i n$.
- $n =_i m$ *if* $m =_i n$.
- $m =_i l$ *if* $m =_i n, n =_i l$.

Definition 4. *We say* a *is reducible if there is a* b *such that* $a \to_i b$. *So* a *is in i-normal form if and only if* a *is not reducible. We say* b *is a normal form of* a *with respect to* $\to_i$ *if* $a \twoheadrightarrow_i b$ *and* b *is not reducible.* a *and* b *are joinable if there is* c *such that* $a \twoheadrightarrow_i c$ *and* $b \twoheadrightarrow_i c$. *An abstract reduction system is strongly normalizing if there are no infinite reduction path.*

2.2 Lambda Encodings

We use $x, y, z, s, n, x_1, x_2, ...$ to denote individual variable, $t, t', a, b, t_1, t_2, ...$ to denote term, $\equiv$ to denote syntactic equality. $[t'/x]t$ to denote substituting the variable x in t for t'. The syntax and reduction for lambda calculus is given as following.

Definition 5 (Lambda Calculus)**.**

Term $t ::= x \mid \lambda x.t \mid t\ t'$

Reduction $(\lambda x.t)t' \to_\beta [t'/x]t$

For example, $(\lambda x.x\ x)(\lambda x.x\ x)$, $\lambda y.y$ are concrete terms in lambda calculus. For a term $\lambda x.t$, we call λ the *binder*, x is *binded* , called *bind variable.* If a variable is not binded, we say it is a *free* variable. We will treat terms up to α-equivalence, meaning, for any term t, one can always rename the binded variables in t. So for example, $\lambda x.x\ x$ is the same as $\lambda y.y\ y$, and $\lambda x.\lambda y.x\ y$ is the same as $\lambda z.\lambda x.z\ x$. $(\lambda x.\lambda y.x\ y)\underline{((\lambda z.z)z_1)} \to_\beta \underline{(\lambda x.\lambda y.x\ y)z_1} \to_\beta \lambda y.z_1\ y$ is a valid reduction sequence in lambda calculus. Note that for reader's convenient we underline the part we are going to carry out the reduction(we will not do this again) and we call the underline term *redex*. For a comprehensive introducton on lambda calculus, we refer to [5].

2.2.1 Church Encoding

Definition 6 (Church Numeral)**.**

$0 \ := \lambda s.\lambda z.z$

$\mathsf{S} \ := \lambda n.\lambda s.\lambda z.s(n\ s\ z)$

So $1 \ := \mathsf{S}\ 0 \equiv (\lambda n.\lambda s.\lambda z.s(n\ s\ z))(\lambda s.\lambda z.z) \to_\beta \lambda s.\lambda z.s((\lambda s.\lambda z.z)s\ z) \to_\beta \lambda s.\lambda z.s\ z$. Note that the last part of above reductions occur underneath the lambda abstractions. Similarly $2 \ := \mathsf{S}\ (\mathsf{S}\ 0) \to_\beta^* \lambda s.\lambda z.s\ s\ z$.

Informally, we can interpret lambda term as both data and function, so instead of thinking Church numeral 2 as data, one can think of it as a higher order function h, which take in a function f and a data a as arguments, then apply the function f to a two times. We define *iterator* $\mathsf{It}\ n\ f\ t \ := n\ f\ t$. So $\mathsf{It}\ 0\ f\ t =_\beta t$ and $\mathsf{It}\ (\mathsf{S}\ u)\ f\ t =_\beta f(\mathsf{It}\ u\ f\ t)$. Then we can use iterator to define $\mathsf{Plus}\ n\ m := \mathsf{It}\ n\ \mathsf{S}\ m$.

2.2.2 Scott Encoding

Definition 7 (Scott Numeral)**.**

$0 \ := \lambda s.\lambda z.z$

$\mathsf{S} \ := \lambda n.\lambda s.\lambda z.s\ n$

We can see $1 \ := \lambda s.\lambda z.(s\ 0)$, $2 \ := \lambda s.\lambda z.(s\ 1)$. We are going to define a notion of *recursor*. We first give a version of the *fix point operator* $\mathsf{Fix} :=$

$\lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$. The reason it is called fix point operator is when it applied to a lambda expression, it give a fix point of that lambda expression(recall informally each lambda expression is both data and function). So

$$\mathsf{Fix}\, g \to_\beta (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x)) \to_\beta g((\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))) =_\beta g\ (\mathsf{Fix}\, g).$$

Now we can define recursor: $\mathsf{Rec} := \mathsf{Fix}\ \lambda r.\lambda n.\lambda f.\lambda v.n\ (\lambda m.f\ (r\ m\ f\ v)\ m)\ v$. We get $\mathsf{Rec}\ 0\ f\ v \twoheadrightarrow_\beta v$ and $\mathsf{Rec}\ (\mathsf{S}\ n)\ f\ v \twoheadrightarrow_\beta f\ (\mathsf{Rec}\ n\ f\ v)\ n$. In a similar fashion, one can define $\mathsf{Plus}\ n\ m := \mathsf{Rec}\ n\ (\lambda x.\lambda y.\mathsf{S}\ x)\ m$.

The predecessor function can be easily defined as $\mathsf{Pred}\ n := \mathsf{Rec}\ n\ (\lambda x.\lambda y.y)\ 0$. It only takes constant time (w.r.t. the number of beta reduction steps) to calculate the predessesor. But this function is tricky to define with Church encoding, one need to first define recursor with iterator, then use recursor to define Pred. To calculate $\mathsf{Pred}\ n$ with Church encoding, one has to perform at least n steps, so it takes linear time [24].

2.2.3 Parigot Encoding

Definition 8 (Parigot Numeral)**.**

$0 := \lambda s.\lambda z.z$

$\mathsf{S} := \lambda n.\lambda s.\lambda z.s\ n\ (n\ s\ z)$

Parigot encoding can be seen as a mixture of Church and Scott encoding, each data contains its own subdata and it support a form of iteration similar to Church encoding. For example, we can define $\mathsf{Pred}\ n := n\ (\lambda x.\lambda y.y)\ 0$ and $\mathsf{Plus}\ n\ m := n\ (\lambda x.\mathsf{S})\ m$. We do not need full recursion to compute with Parigot numerals and we can retrieve subdata in constant time.

2.3 Confluence

Definition 9. *Given an abstract reduction system $(\mathcal{A}, \{\rightarrow_i\}_{i\in\mathcal{I}})$, let $\rightarrow$ denote $\bigcup_{i\in\mathcal{I}} \rightarrow_i$, let $=$ denote the equivalence relation generated by $\rightarrow$.*

- *Confluence: For any $a, b, c \in \mathcal{A}$, if $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, then there exist $d \in \mathcal{A}$ such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.*

- *Church-Rosser: For any $a, b \in \mathcal{A}$, if $a = b$, then there is a $c \in \mathcal{A}$ such that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$.*

The two properties above can be expressed by following diagrams:

$$\begin{tikzcd}
& a \arrow[dl, two heads] \arrow[dr, two heads] & \\
b \arrow[dr, dashed, two heads] & & c \arrow[dl, dashed, two heads] \\
& d &
\end{tikzcd}
\qquad
\begin{tikzcd}
a \arrow[r, phantom, "="] \arrow[dr, dashed, two heads] & {} & b \arrow[dl, dashed, two heads] \\
& c &
\end{tikzcd}$$

Lemma 1. *An abstract reduction system $\mathcal{R}$ is confluent iff it is Church-Rosser.*

Proof. Assume the same notation as defintion 9.

"$\Leftarrow$": Assume $\mathcal{R}$ is Church-Rosser. For any $a, b, c \in \mathcal{A}$, if $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, then this means $b = c$. By Church-Rosser, there is a $d \in \mathcal{A}$, such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

"$\Rightarrow$": Assume $\mathcal{R}$ is Confluent. For any $a, b \in \mathcal{A}$, if $a = b$, then we show there is a $c \in \mathcal{A}$ such that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$ by induction on the generation of $a = b$:

If $a \twoheadrightarrow b \Rightarrow a = b$, then let c be b.

If $b = a \Rightarrow a = b$, by induction, there is a c such that $b \twoheadrightarrow c$ and $a \twoheadrightarrow c$.

If $a = d, d = b \Rightarrow a = b$, by induction there is a c_1 such that $a \twoheadrightarrow c_1$ and $d \twoheadrightarrow c_1$; there is a c_2 such that $d \twoheadrightarrow c_2$ and $b \twoheadrightarrow c_2$. So now we get $d \twoheadrightarrow c_1$ and $d \twoheadrightarrow c_2$, by confluence, we have a c such that $c_1 \twoheadrightarrow c$ and $c_2 \twoheadrightarrow c$. So $a \twoheadrightarrow c_1 \twoheadrightarrow c$ and $b \twoheadrightarrow c_2 \twoheadrightarrow c$. This process is illustrated by the following diagram:

$$\begin{tikzcd}
a \arrow[r, phantom, "="] \arrow[dr, dashed] & d \arrow[r, phantom, "="] \arrow[d, dashed] \arrow[dr, dashed] & b \arrow[d, dashed] \\
& c_1 \arrow[d, dashed] & c_2 \arrow[dl, dashed] \\
& c &
\end{tikzcd}$$

□

The definition of $=$ depends on $\twoheadrightarrow$, the definition of $\twoheadrightarrow$ depends on $\rightarrow$, confluence is often easier to prove compare to Church-Rosser, in the sense that it is easier to anaylze $\twoheadrightarrow$ compare to $=$. Now let us see some consequences of confluence.

Corollary 1. *If $\mathcal{R}$ is confluent, then every element in $\mathcal{A}$ has at most one normal form.*

Proof. Assume $a \in \mathcal{A}$, b, c are two diferent normal forms for a. So we have $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, by confluence, there exist a d such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$. But b, c are normal form, this implies b and c are the same as d, which contradicts that they are two different normal form. □

Definition 10. *For an abstract reduction system $\mathcal{R}$, it is trivial if for any $a, b \in \mathcal{A}$, $a = b$.*

Corollary 2. *If $\mathcal{R}$ is confluent and there are at least two different normal forms, then $\mathcal{R}$ is not trivial.*

2.4 Tait-Martin Löf's Method

We want to show lambda calculus as an abstract reduction system is confluent. We present a method of proving confluence in abstract reduction system, which is due to W. Tait and P. Martin-Löf(reported in [5]). Then we show how we can apply this method to show lambda calculus is confluent.

Definition 11 (Diamond Property). *Given an abstract reduction system* $(\mathcal{A}, \{\rightarrow_i\}_{i\in\mathcal{I}})$*, it has diamond property if:*

For any $a, b, c \in \mathcal{A}$*, if* $a \rightarrow b$ *and* $a \rightarrow c$*, then there exist* $d \in \mathcal{A}$ *such that* $b \rightarrow d$ *and* $c \rightarrow d$*.*

$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b \arrow[dr, dashed] & & c \arrow[dl, dashed] \\ & d & \end{tikzcd}$$

Lemma 2. *If* $\mathcal{R}$ *has diamond property, then it is confluent.*

Proof. By simple diagam chasing suggested below:

$$\begin{tikzcd} & & a \arrow[dl] \arrow[dr] & & \\ & c_1 \arrow[dl] \arrow[dr, dashed] & & c_2 \arrow[dl, dashed] \arrow[dr] & \\ e \arrow[dr, dashed] & & d \arrow[dl, dashed] \arrow[dr, dashed] & & b \arrow[dl, dashed] \\ & c_1 \arrow[dr, dashed] & & c_2 \arrow[dl, dashed] & \\ & & c & & \end{tikzcd}$$

□

Lemma 3. *If exist some $\to_i$, $\to \subseteq \to_i \subseteq \twoheadrightarrow$ and $\to_i$ satisfies diamond property, then $\to$ is confluent.*

Proof. Since $\to \subseteq \to_i \subseteq \twoheadrightarrow$ implies $\twoheadrightarrow \subseteq \twoheadrightarrow_i \subseteq \twoheadrightarrow$, so $\twoheadrightarrow_i = \twoheadrightarrow$. And the diamond property of $\to_i$ implies $\to_i$ is confluence, thus implies the confluence of $\to$. □

Sometimes $\to$ may not satisfy diamond property, then one can look for the possibility to construct an intermediate reduction $\to_i$ such that it has diamond property. That is exactly what we will do for lambda calculus. Beta reduction itself does not satsify diamond property, for example, $(\lambda x.((\lambda u.u)\ v)\ ((\lambda y.y\ y)\ z) \to_\beta (\lambda x.((\lambda u.u)\ v))\ (z\ z)$ and $(\lambda x.((\lambda u.u)\ v)\ ((\lambda y.y\ y)\ z) \to_\beta (\lambda u.u)\ v$. And one can not join $(\lambda u.u)\ v$ and $(\lambda x.((\lambda u.u)\ v))\ (z\ z)$ in one step. But one can see they are still joinable, but not joinable in one step. This leads to the notion of parallel reduciton.

Definition 12 (Parallel Reduction)**.**

$$\frac{}{t \Rightarrow_\beta t} \quad \frac{t \Rightarrow_\beta t'}{\lambda x.t \Rightarrow_\beta \lambda x.t'} \quad \frac{t_1 \Rightarrow_\beta t_1' \quad t_2 \Rightarrow_\beta t_2'}{t_1 t_2 \Rightarrow_\beta t_1' t_2'} \quad \frac{t_1 \Rightarrow_\beta t_1' \quad t_2 \Rightarrow_\beta t_2'}{(\lambda x.t_1)t_2 \Rightarrow_\beta [t_2'/x]t_1'}$$

Intuitively, parallel reduction allows us to contract many beta redex(or not contracting at all) in one step, under this notion of one step reduction, we can obtain diamond property for $\Rightarrow_\beta$.

Lemma 4. *If $t_1 \Rightarrow_\beta t_1'$ and $t_2 \Rightarrow_\beta t_2'$, then $[t_2/x]t_1 \Rightarrow_\beta [t_2'/x]t_1'$.*

Proof. By induction on the derivation of $t_1 \Rightarrow_\beta t_1'$. We will not prove this here. □

Lemma 5. *$\Rightarrow_\beta$ satisfies diamond property.*

Proof. Assume $t \Rightarrow_\beta t_1$ and $t \Rightarrow_\beta t_2$, we need to show there exists a t_3 such that $t_1 \Rightarrow_\beta t_3$ and $t_2 \Rightarrow_\beta t_3$. We prove this by induction on the derivation of $t \Rightarrow_\beta t_1$.

- **Case**: $\dfrac{}{t \Rightarrow_\beta t}$

  Simply let t_3 be t.

- **Case**: $\dfrac{t' \Rightarrow_\beta t''}{\lambda x.t' \Rightarrow_\beta \lambda x.t''}$

  In this case t is of the form $\lambda x.t'$, where $t' \Rightarrow_\beta t''$; t_1 is of the form $\lambda x.t''$. t_2 must be of the form $\lambda x.t'''$, where $t' \Rightarrow_\beta t'''$. Thus by induction, we have a t_3' such that $t'' \Rightarrow_\beta t_3'$ and $t''' \Rightarrow_\beta t_3'$. Thus let t_3 be $\lambda x.t_3'$, we get $t_1 \equiv \lambda x.t'' \Rightarrow_\beta \lambda x.t_3' \equiv t_3$ and $t_2 \equiv \lambda x.t''' \Rightarrow_\beta \lambda x.t_3' \equiv t_3$.

- **Case**: $\dfrac{t_4 \Rightarrow_\beta t_4' \quad t_5 \Rightarrow_\beta t_5'}{(\lambda x.t_4)t_5 \Rightarrow_\beta [t_4'/x]t_5'}$

  In this case t is of the form $(\lambda x.t_4)t_5$, t_1 is of the form $[t_5'/x]t_4'$, $t_4 \Rightarrow_\beta t_4'$ and $t_5 \Rightarrow_\beta t_5'$.

  If t_2 is of the form $(\lambda x.t_4'')t_5''$, where $t_4 \Rightarrow_\beta t_4''$ and $t_5 \Rightarrow_\beta t_5''$. Thus by induction, we have a t_6 such that $t_5'' \Rightarrow_\beta t_6$ and $t_5' \Rightarrow_\beta t_6$. And same by induction, there is a t_7 such that $t_4'' \Rightarrow_\beta t_7$ and $t_4' \Rightarrow_\beta t_7$. Thus let t_3 be $[t_6/x]t_7$, we get $t_1 \equiv [t_5'/x]t_4' \Rightarrow_\beta [t_6/x]t_7 \equiv t_3$(by lemma 4) and $t_2 \equiv (\lambda x.t_4'')t_5'' \Rightarrow_\beta [t_6/x]t_7 \equiv t_3$.

  If t_2 is of the form $[t_5''/x]t_4''$, where $t_4 \Rightarrow_\beta t_4''$ and $t_5 \Rightarrow_\beta t_5''$. Thus by induction, we have a t_6 such that $t_5'' \Rightarrow_\beta t_6$ and $t_5' \Rightarrow_\beta t_6$. And same by induction, there is a t_7 such that $t_4'' \Rightarrow_\beta t_7$ and $t_4' \Rightarrow_\beta t_7$. Thus let t_3 be $[t_6/x]t_7$, by lemma 4, we get $t_1 \equiv [t_5'/x]t_4' \Rightarrow_\beta [t_6/x]t_7 \equiv t_3$ and $t_2 \equiv [t_5''/x]t_4'' \Rightarrow_\beta [t_6/x]t_7 \equiv t_3$.

- **Case**: $\dfrac{t_4 \Rightarrow_\beta t'_4 \quad t_5 \Rightarrow_\beta t'_5}{t_4 t_5 \Rightarrow_\beta t'_4 t'_5}$

  Similar to the arguments above.

□

Lemma 6. $\to_\beta \subseteq \Rightarrow_\beta \subseteq \twoheadrightarrow_\beta$.

Theorem 1. $\to_\beta$ *reduction is confluent.*

Proof. By lemma 3, lemma 5 and lemma 6. □

2.5 Hardin's Interpretation Method

Sometimes it is inevitable to deal with reduction systems that contains more than one reduction, for example, $(\Lambda, \{\to_\beta, \to_\eta\})$. Confluence problem for this kind of system require some nontrivial efforts to prove. Hardin's interpretion method [25] provide a way to deal with some of those reduction systems.

Lemma 7 (Interpretation lemma)**.** *Let* $\to$ *be* $\to_1 \cup \to_2$, $\to_1$ *being confluent and strongly normalizing. We denote by* $\nu(a)$ *the* $\to_1$*-normal form of* a*. Suppose that there is some relation* $\to_i$ *on* $\to_1$ *normal forms satisfying:*

$\to_i \subseteq \twoheadrightarrow$, *and* $a \to_2 b$ *implies* $\nu(a) \twoheadrightarrow_i \nu(b)$ (†)

Then the confluence of $\to_i$ *implies the confluence of* $\to$.

Proof. So suppose $\to_i$ is confluent. If $a \twoheadrightarrow a'$ and $a \twoheadrightarrow a''$. So by (†), $\nu(a) \twoheadrightarrow_i \nu(a')$ and $\nu(a) \twoheadrightarrow_i \nu(a'')$. Notice that $t \to_1^* t'$ implies $\nu(t) = \nu(t')$(By confluence and strong normalizing of $\rightarrow_1$). By confluence of $\rightarrow_i$, there exists b such that $\nu(a')\twoheadrightarrow_i b$ and $\nu(a'')\twoheadrightarrow_i b$. Since $\rightarrow_i, \rightarrow_1 \subseteq \twoheadrightarrow$, we got $a' \twoheadrightarrow \nu(a') \twoheadrightarrow b$ and $a'' \twoheadrightarrow \nu(a'') \twoheadrightarrow b$. Hence $\rightarrow$ is confluent.

$$\begin{tikzcd}
 & & & & a \arrow[dllll, two heads] \arrow[d, dashed, two heads, "1"'] \arrow[drrrr, two heads] & & & & \\
a' \arrow[drr, dashed, two heads, "1"] & & & & \nu(a) \arrow[dll, dashed, two heads, "i"'] \arrow[drr, dashed, two heads, "i"] & & & & a'' \arrow[dll, dashed, two heads, "1"'] \\
 & & \nu(a') \arrow[drr, dashed, two heads, "i"] & & & & \nu(a'') \arrow[dll, dashed, two heads, "i"'] & & \\
 & & & & b & & & &
\end{tikzcd}$$

$\square$

Hardin's method reduce the confluence problem of $\rightarrow_1 \cup \rightarrow_2$ to $\rightarrow_i$, given the confluence and strong normalizing of $\rightarrow_1$, this make it possible to apply Tait-Martin-Löf's method to prove confluence of $\rightarrow_i$.

CHAPTER 3

CONFLUENCE OF LAMBDA-MU CALCULUS

In this Chapter, we will investigate the confluence problem of extending lambda calculus with local definitions, we called it $\lambda\mu$ calculus, the μ represents the usual let-rec binding availables in functional programming languages. It is desirable to know the confluence property of $\lambda\mu$ since it may be needed to prove type preservation for the type systems based on $\lambda\mu$ and it implies the equality reasoning is consistent. We give the formulation of $\lambda\mu$ first (Section 3.1). We discuss why traditional approaches to confluence fail on $\lambda\mu$ in Section 3.2. Finally, we show how to use interpretation method to prove the confluence for a restrictive version of $\lambda\mu$ calculus (we called it *local* lambda-mu calculus) in Section 3.3.

3.1 Lambda-Mu Calculus

Definition 13 (Syntax)**.**

Terms $t \ ::= \ x \mid \lambda x.t \mid tt' \mid \mu t$

Local Definitions/Closures $\mu \ ::= \{x_i \mapsto t_i\}_{i\in\mathcal{I}}$

Definition 14 (Free Variables)**.**

$\mathrm{FV}(x) := x.$

$\mathrm{FV}(\lambda x.t) := \mathrm{FV}(t)/x.$

$\mathrm{FV}(t\ t') := \mathrm{FV}(t) \cup \mathrm{FV}(t')$

$\mathrm{FV}(\mu t) := (\mathrm{FV}(t) - \mathrm{dom}(\mu)) \cup \mathrm{FV}(\mu)$

$\mathrm{FV}(\{x_i \mapsto t_i\}_{i\in\mathcal{I}}) := (\bigcup_{i\in\mathcal{I}} \mathrm{FV}(t_i)) - \{x_i\}_{i\in\mathcal{I}}$

Definition 15 (Beta-Reductions). $\boxed{t \to_\beta t'}$

$$\frac{}{(\lambda x.t)t' \to_\beta [t'/x]t} \quad \frac{(x_i \mapsto t_i) \in \mu}{\mu x_i \to_\beta \mu t_i} \quad \frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'} \quad \frac{t \to_\beta t''}{tt' \to_\beta t''t'}$$

$$\frac{t' \to_\beta t''}{t\ t' \to_\beta t\ t''} \quad \frac{t \to_\beta t'}{\mu t \to_\beta \mu t'} \quad \frac{\mu \to_\beta \mu'}{\mu t \to_\beta \mu' t}$$

Note that $\mu \to_\beta \mu'$ is a shorthand for there is exactly one $x_i \mapsto t_i \in \mu$ and $t_i \to_\beta t'_i$, and μ' is same as μ except $x_i \mapsto t'_i \in \mu'$. Similarly, we have shorthand for $\mu \to_\mu \mu'$.

Definition 16 (Mu-Reductions). $\boxed{t \to_\mu t'}$

$$\frac{\mathrm{dom}(\mu)\#\mathrm{FV}(t)}{\mu t \to_\mu t} \quad \frac{}{\mu(\lambda x.t) \to_\mu \lambda x.\mu t} \quad \frac{}{\mu(t_1 t_2) \to_\mu (\mu t_1)(\mu t_2)}$$

$$\frac{t \to_\mu t'}{\lambda x.t \to_\mu \lambda x.t'} \quad \frac{t' \to_\mu t''}{t\ t' \to_\mu t\ t''} \quad \frac{t \to_\mu t''}{t\ t' \to_\mu t''\ t'}$$

$$\frac{t \to_\mu t'}{\mu t \to_\mu \mu t'} \quad \frac{\mu \to_\mu \mu'}{\mu t \to_\mu \mu' t}$$

Mutual substitutions within the local definition is not possible in $\lambda\mu$, because of Ariola and Klop [4]'s non-confluence example:

$$\{\delta \mapsto \underline{\lambda y.\alpha(\mathsf{S}y)}, \alpha \mapsto \lambda x.\delta(\mathsf{S}x)\}\alpha \to \{\delta \mapsto \lambda y.\delta\mathsf{S}(\mathsf{S}y), \alpha \mapsto \lambda x.\delta(\mathsf{S}x)\}\alpha$$

$$\{\delta \mapsto \lambda y.\alpha(\mathsf{S}y), \alpha \mapsto \underline{\lambda x.\delta(\mathsf{S}x)}\}\alpha \to \{\delta \mapsto \lambda y.\alpha(\mathsf{S}y), \alpha \mapsto \lambda x.\alpha\mathsf{S}(\mathsf{S}x)\}\alpha.$$

It seems natural to allow mutual substitutions. We consider mutal substitutions overly eager. Because in the above non-confluence example, only α is being used, namely, occurs in the body. So there is no need to reduce the α in the definiens of the δ if one is "lazy" enough.

Another possible formulation of mu-reduction is $(\mu t\ \mu t') \to \mu(t\ t')$ instead of pushing μ inside of a term as we do. The potential drawback is in the case where $(\mu(\lambda x.t))t'$, where there is no μ inside t', it is now a stuck term. One could add another rule to repair this situation: $(\mu(\lambda x.t))t' \to \mu([t'/x]t)$. Then one would need another rule to deal with the case where $(\mu_2\mu_1(\lambda x.t))t'$ etc.

3.2 A Fail Attempt to Prove Confluence of Lambda-Mu Calculus

We want to point out that directly applying Tait-Martin Löf's method (Section 2.4, Chapter 2) will not work for lambda-mu calculus. For example, let $\Rightarrow$ be a direct parallelization of $\to_\beta$ and $\to_\mu$. Then

$\mu((\lambda x.x)\ t) \Rightarrow \mu' t'$, where $\mu \Rightarrow \mu', t \Rightarrow t'$.

$\mu((\lambda x.x)\ t) \Rightarrow (\mu'(\lambda x.x))\ (\mu' t)$, where $\mu \Rightarrow \mu'$.

We can not bring back $(\mu'(\lambda x.x))\ (\mu' t)$ and $\mu' t'$ in one $\Rightarrow$ step. Thus the $\Rightarrow$ does not have the diamond property.

Since we know that the $\to_\mu$ reduction is convergent, then we would hope to use Hardin's interpretation lemma to reduce the confluence proof of $\to_\beta \cup \to_\mu$ to $\to_{\beta\mu}$, which is a reduction defined on *mu-normal* term, and then apply Tait-Martin Löf's method to show confluence of $\to_{\beta\mu}$. We fail on the second step, namely, applying Tait-Martin Löf's method to show confluence of $\to_{\beta\mu}$. We will introduce several definitions before we discuss the reason we fail.

Lemma 8. *$\to_\mu$ is strongly normalizing and confluent.*

Proof. The number of μ-redex is decreasing by the $\to_\mu$-reduction. We can use local confluence to prove confluence. $\square$

Definition 17 (μ-Normal Forms)**.**

Normal Term $n \ ::= \ x \mid \vec{\rho}x_i \mid \lambda x.n \mid n\ n'$, *where* $x_i \in \text{dom}(\rho_i)$.

Normal Local Definitions $\rho \ ::= \{x_i \mapsto n_i\}_{i \in \mathcal{I}}$

We use $\vec{\mu}t$ to denote $\mu_1 ... \mu_n t$, $\vec{\rho}t$ denote $\rho_1 ... \rho_n t$.

Definition 18 (μ-Normalize Funciton)**.** *We define function ν that maps a term to its μ-normal form.*

$$\begin{array}{ll}
\nu(x) \ := \ x & \nu(\lambda y.t) \ := \ \lambda y.\nu(t) \\
\nu(t_1 t_2) \ := \ \nu(t_1)\nu(t_2) & \nu(\vec{\mu}y) \ := y \textit{ if } y \notin \text{dom}(\vec{\mu}). \\
\nu(\vec{\mu}y) \ := \nu(\vec{\mu})y \textit{ if } y \in \text{dom}(\mu_i). & \nu(\vec{\mu}(tt')) \ := \nu(\vec{\mu}t)\nu(\vec{\mu}t') \\
\nu(\vec{\mu}(\lambda x.t)) \ := \lambda x.\nu(\vec{\mu}t). & \nu(x \mapsto t, \mu) \ := x \mapsto \nu(t), \nu(\mu).
\end{array}$$

Definition 19 (β Reduction on μ-normal Forms)**.** $\boxed{n \rightarrow_{\beta\mu} n'}$

$$\frac{n \rightarrow_\beta t}{n \rightarrow_{\beta\mu} \nu(t)} \quad \frac{n \rightarrow_{\beta\mu} n'}{\lambda x.n \rightarrow_{\beta\mu} \lambda x.n'} \quad \frac{n' \rightarrow_{\beta\mu} n''}{n\ n' \rightarrow_{\beta\mu} n\ n''} \quad \frac{n \rightarrow_{\beta\mu} n''}{n\ n' \rightarrow_{\beta\mu} n''\ n'}$$

Intuitively, $\rightarrow_{\beta\mu}$ first $\rightarrow_\beta$ reduce a term and then apply the ν function to the contractum.

Definition 20 (Parallelization)**.** $\boxed{n \Rightarrow_{\beta\mu} n'}$

$$\frac{}{n \Rightarrow_{\beta\mu} n} \qquad \frac{(x_i \mapsto n_i) \in \rho_i}{\vec{\rho}x_i \Rightarrow_{\beta\mu} \nu(\vec{\rho}n_i)} \qquad \frac{n_1 \Rightarrow_{\beta\mu} n_1' \quad n_2 \Rightarrow_{\beta\mu} n_2'}{(\lambda x.n_1)n_2 \Rightarrow_{\beta\mu} \nu([n_1'/x]n_2')}$$

$$\frac{n \Rightarrow_{\beta\mu} n'}{\lambda x.n \Rightarrow_{\beta\mu} \lambda x.n'} \quad \frac{n' \Rightarrow_{\beta\mu} n''' \quad n \Rightarrow_{\beta\mu} n''}{n\ n' \Rightarrow_{\beta\mu} n''\ n'''} \quad \frac{\vec{\rho} \Rightarrow_{\beta\mu} \vec{\rho'}}{\vec{\rho}x_i \Rightarrow_{\beta\mu} \vec{\rho'}x_i}$$

Note that $\vec{\rho} \Rightarrow_{\beta\mu} \vec{\rho'}$ is a shorthand for for all ρ_i, and for all $x_i \mapsto n_i \in \rho_i$, we have $n_i \Rightarrow_{\beta\mu} n_i'$, and ρ_i' is consisted of $x_i \mapsto n_i'$.

The next step would be to show $\Rightarrow_{\beta\mu}$ has diamond property so that we can conclude the confluence of $\rightarrow_{\beta\mu}$. However, $\Rightarrow_{\beta\mu}$ does not have the diamond property due to the following counter-example:

Let μ denote $\{x \mapsto (\lambda y.y)\ z, z \mapsto \lambda q.q\}$.

$\{x \mapsto (\lambda y.y)\ z, z \mapsto \lambda q.q\}\ x \Rightarrow_{\beta\mu} \{x \mapsto z, z \mapsto \lambda q.q\}\ x$

$\{x \mapsto (\lambda y.y)\ z, z \mapsto \lambda q.q\}\ x \Rightarrow_{\beta\mu} (\lambda y.\mu y)\ (\mu z)$.

We can not join $(\lambda y.\mu y)\ (\mu z)$ and $\{x \mapsto z, z \mapsto \lambda q.q\}\ x$ in one $\Rightarrow_{\beta\mu}$ step. So at this point even though intuitive it seems like lambda-mu calculus is confluent, we have not found an adequate proof yet.

3.3 Confluence of Local Lambda-Mu Calculus

In this section we are going to prove confluence of a restrictive version of lambda-mu calculus, namely, *local* lambda-mu calculus. For local lambda-mu, given $\{x_i \mapsto t_i\}_{i \in N} t$, we require for any $1 \leq i \leq n$, the set of free variables of t_i, $\mathrm{FV}(t_i) \subseteq \mathrm{dom}(\mu) = \{x_1, ..., x_n\}$ and we do not allow reduction, definitional substitution, substitution inside the definitions.

Definition 21 (Beta-Reductions). $\boxed{t \rightarrow_\beta t'}$

$$\frac{}{(\lambda x.t)t' \rightarrow_\beta [t'/x]t} \qquad \frac{(x_i \mapsto t_i) \in \mu}{\mu x_i \rightarrow_\beta \mu t_i} \qquad \frac{t \rightarrow_\beta t'}{\lambda x.t \rightarrow_\beta \lambda x.t'}$$

$$\frac{t \rightarrow_\beta t''}{tt' \rightarrow_\beta t''t'} \qquad \frac{t' \rightarrow_\beta t''}{tt' \rightarrow_\beta tt''} \qquad \frac{t \rightarrow_\beta t'}{\mu t \rightarrow_\beta \mu t'}$$

Definition 22 (Mu-Reductions). $\boxed{t \rightarrow_\mu t'}$

$$\frac{\mathrm{dom}(\mu)\#\mathrm{FV}(t)}{\mu t \to_\mu t} \qquad \frac{}{\mu(\lambda x.t) \to_\mu \lambda x.\mu t} \qquad \frac{}{\mu(t_1 t_2) \to_\mu (\mu t_1)(\mu t_2)}$$

$$\frac{t \to_\mu t'}{\lambda x.t \to_\mu \lambda x.t'} \qquad \frac{t' \to_\mu t''}{tt' \to_\mu tt''} \qquad \frac{t \to_\mu t''}{tt' \to_\mu t''t'}$$

$$\frac{t \to_\mu t'}{\mu t \to_\mu \mu t'}$$

Lemma 9. $\to_\mu$ *is strongly normalizing and confluent.*

Definition 23 (μ-Normal Forms)**.**

$n \ ::= \ x \mid \mu x_i \mid \lambda x.n \mid n\ n'$

We require $x_i \in \mathrm{dom}(\mu)$.

Definition 24 (μ-Normalize Funciton)**.**

$$\begin{array}{ll} \nu(x) \ := \ x & \nu(\lambda y.t) \ := \ \lambda y.\nu(t) \\ \nu(t_1 t_2) \ := \ \nu(t_1)\nu(t_2) & \nu(\vec{\mu}y) \ := y \ \textit{if}\ y \notin \mathrm{dom}(\vec{\mu}). \\ \nu(\vec{\mu}y) \ := \mu_i y \ \textit{if}\ y \in \mathrm{dom}(\mu_i). & \nu(\vec{\mu}(tt')) \ := \nu(\vec{\mu}t)\nu(\vec{\mu}t') \\ \nu(\vec{\mu}(\lambda x.t)) \ := \lambda x.\nu(\vec{\mu}t). & \end{array}$$

Definition 25 (β Reduction on μ-normal Forms)**.**

$$\frac{n \to_\beta t}{n \to_{\beta\mu} \nu(t)} \qquad \frac{n \to_{\beta\mu} n'}{\lambda x.n \to_{\beta\mu} \lambda x.n'} \qquad \frac{n' \to_{\beta\mu} n''}{nn' \to_{\beta\mu} nn''} \qquad \frac{n \to_{\beta\mu} n''}{nn' \to_{\beta\mu} n''n'}$$

Definition 26 (Parallelization)**.**

$$\frac{}{n \Rightarrow_{\beta\mu} n} \qquad \frac{(x_i \mapsto t_i) \in \mu}{\mu x_i \Rightarrow_{\beta\mu} \nu(\mu t_i)} \qquad \frac{n_1 \Rightarrow_{\beta\mu} n_1' \quad n_2 \Rightarrow_{\beta\mu} n_2'}{(\lambda x.n_1)n_2 \Rightarrow_{\beta\mu} \nu([n_1'/x]n_2')}$$

$$\frac{n \Rightarrow_{\beta\mu} n'}{\lambda x.n \Rightarrow_{\beta\mu} \lambda x.n'} \qquad \frac{n' \Rightarrow_{\beta\mu} n''' \quad n \Rightarrow_{\beta\mu} n''}{nn' \Rightarrow_{\beta\mu} n''n'''}$$

Lemma 10. $\to_{\beta\mu} \subseteq \Rightarrow_{\beta\mu} \subseteq \to_{\beta\mu}^*$.

Lemma 11. *If* $n_2 \Rightarrow_{\beta\mu} n_2'$*, then* $\nu([n_2/x]n_1) \Rightarrow_{\beta\mu} \nu([n_2'/x]n_1)$.

Proof. By induction on the structure of n_1.

Base Cases: $n_1 = x$, $n_1 = \mu x_i$, Obvious.

Step Case: $n_1 = \lambda y.n$. We have $\nu(\lambda y.[n_2/x]n) \equiv \lambda y.\nu([n_2/x]n) \overset{IH}{\Rightarrow_{\beta\mu}} \lambda y.\nu([n_2'/x]n) \equiv \nu(\lambda y.[n_2'/x]n)$.

Step Case: $n_1 = nn'$. We have $\nu([n_2/x]n[n_2/x]n') \equiv \nu([n_2/x]n)\nu([n_2/x]n') \overset{IH}{\Rightarrow_{\beta\mu}} \nu([n_2'/x]n)\nu([n_2'/x]n') \equiv \nu([n_2'/x]n[n_2'/x]n)$.

□

We use $\overrightarrow{\dot{\mu}}$ to denote zero or more μs.

Lemma 12. *$\nu(\nu(t)) \equiv \nu(t)$ and $\nu([\nu(t_1)/y]\nu(t_2)) \equiv \nu([t_1/y]t_2)$.*

Proof. We only prove the second equality here. We identify t_2 as $\overrightarrow{\dot{\mu}_1}t_2'$, where t_2' does not contains any closure at head position. We proceed by induction on the structure of t_2':

Base Cases: For $t_2' = x$, we use $\nu(\nu(t)) \equiv \nu(t)$.

Step Cases: If $t_2' = \lambda x.t_2''$, then

$$\nu(\overrightarrow{\dot{\mu}_1}(\lambda x.[t_1/y]t_2'')) \equiv \lambda x.\nu(\overrightarrow{\dot{\mu}_1}([t_1/y]t_2'')) \equiv \lambda x.\nu(\overrightarrow{\dot{\mu}_1}\overrightarrow{\dot{\mu}_2}([t_1/y]t_2''')),$$

where t_2'' as $\overrightarrow{\dot{\mu}_2}t_2'''$ and t_2''' does not have any closure at head position. Since t_2''' is structurally smaller than $\lambda x.t_2''$, by IH, $\nu(\overrightarrow{\dot{\mu}_1}\overrightarrow{\dot{\mu}_2}([t_1/y]t_2''')) \equiv \nu([t_1/y](\overrightarrow{\dot{\mu}_1}\overrightarrow{\dot{\mu}_2}t_2''')) \equiv \nu([\nu(t_1)/y]\nu(\overrightarrow{\dot{\mu}_1}\overrightarrow{\dot{\mu}_2}t_2'''))$. Thus $\lambda x.\nu(\overrightarrow{\dot{\mu}_1}\overrightarrow{\dot{\mu}_2}([t_1/y]t_2''')) \equiv \lambda x.\nu([\nu(t_1)/y]\nu(\overrightarrow{\dot{\mu}_1}\overrightarrow{\dot{\mu}_2}t_2'''))$. So $\nu([t_1/y]\overrightarrow{\dot{\mu}_1}(\lambda x.t_2'')) \equiv \nu([\nu(t_1)/y]\nu(\lambda x.\overrightarrow{\dot{\mu}_1}\overrightarrow{\dot{\mu}_2}t_2''')) \equiv \nu([\nu(t_1)/y]\nu(\lambda x.\overrightarrow{\dot{\mu}_1}t_2'')) \equiv \nu([\nu(t_1)/y]\nu(\overrightarrow{\dot{\mu}_1}(\lambda x.t_2'')))$

For $t_2' = t_a t_b$, we can argue similarly as above.

□

Lemma 13. *If $n_1 \Rightarrow_{\beta\mu} n_1'$ and $n_2 \Rightarrow_{\beta\mu} n_2'$, then $\nu([n_2/x]n_1) \Rightarrow_{\beta\mu} \nu([n_2'/x]n_1')$.*

Proof. We prove this by induction on the derivation of $n_1 \Rightarrow_{\beta\mu} n_1'$.

- **Base Case:** $\overline{n \Rightarrow_{\beta\mu} n}$

  By the lemma 11.

- **Base Case:** $\dfrac{x_i \mapsto t_i \in \mu}{\mu x_i \Rightarrow_{\beta\mu} \nu(\mu t_i)}$

  Because $y \notin \mathrm{FV}(\mu x_i)$ and μ is local.

- **Step Case:** $\dfrac{n_a \Rightarrow_{\beta\mu} n_a' \quad n_b \Rightarrow_{\beta\mu} n_b'}{(\lambda x.n_a)n_b \Rightarrow_{\beta\mu} \nu([n_a'/x]n_b')}$

  We have $\nu((\lambda x.[n_2/y]n_a)[n_2/y]n_b) \equiv (\lambda x.\nu([n_2/y]n_a))\nu([n_2/y]n_b)$

  $\overset{IH}{\Rightarrow}_{\beta\mu} \nu([\nu([n_2'/y]n_b')/x]\nu([n_2'/y]n_a')) \equiv \nu([n_2'/y]([n_b'/x]n_a'))$. The last equality is by lemma 12.

- **Step Case:** $\dfrac{n \Rightarrow_{\beta\mu} n'}{\lambda x.n \Rightarrow_{\beta\mu} \lambda x.n'}$

  We have $\nu(\lambda x.[n_2/y]n) \equiv \lambda x.\nu([n_2/y]n) \overset{IH}{\Rightarrow}_{\beta\mu} \lambda x.\nu([n_2'/y]n') \equiv \nu(\lambda x.[n_2'/y]n')$

- **Step Case:** $\dfrac{n_a \Rightarrow_{\beta\mu} n_a' \quad n_b \Rightarrow_{\beta\mu} n_b'}{n_a n_b \Rightarrow_{\beta\mu} n_a' n_b'}$

  We have $\nu([n_2/y]n_a[n_2/y]n_b) \equiv \nu([n_2/y]n_a)\nu([n_2/y]n_b)$

  $\overset{IH}{\Rightarrow}_{\beta\mu} \nu([n_2'/y]n_a')\nu([n_2'/y]n_b') \equiv \nu([n_2'/y](n_a'n_b'))$.

□

Lemma 14 (Diamond Property). *If $n \Rightarrow_{\beta\mu} n'$ and $n \Rightarrow_{\beta\mu} n''$, then there exist n''' such that $n'' \Rightarrow_{\beta\mu} n'''$ and $n' \Rightarrow_{\beta\mu} n'''$. So $\to_{\beta\mu}$ is confluent.*

Proof. By induction on the derivation of $n \Rightarrow_{\beta\mu} n'$.

- **Base Case:** $\overline{n \Rightarrow_{\beta\mu} n}$ and $\overline{\mu x_i \Rightarrow_{\beta\mu} \nu(\mu t_i)}$

  Obvious.

- **Step Case:** $\dfrac{n_1 \Rightarrow_{\beta\mu} n_1' \quad n_2 \Rightarrow_{\beta\mu} n_2'}{(\lambda x.n_1)n_2 \Rightarrow_{\beta\mu} \nu([n_1'/x]n_2')}$

  Suppose $(\lambda x.n_1)n_2 \Rightarrow_{\beta\mu} (\lambda x.n_1'')n_2''$, where $n_1 \Rightarrow_{\beta\mu} n_1''$ and $n_2 \Rightarrow_{\beta\mu} n_2''$. By lemma 13 and IH, we have $\nu([n_1'/x]n_2') \Rightarrow_{\beta\mu} \nu([n_1'''/x]n_2''')$. We also have $(\lambda x.n_1'')n_2'' \Rightarrow_{\beta\mu} \nu([n_1'''/x]n_2''')$, where $n_1'' \Rightarrow_{\beta\mu} n_1'''$ and $n_1' \Rightarrow_{\beta\mu} n_1'''$ and $n_2' \Rightarrow_{\beta\mu} n_2'''$ and $n_2' \Rightarrow_{\beta\mu} n_2'''$.

  Suppose $(\lambda x.n_1)n_2 \Rightarrow_{\beta\mu} \nu([n_2''/x]n_1'')$, where $n_1 \Rightarrow_{\beta\mu} n_1''$ and $n_2 \Rightarrow_{\beta\mu} n_2''$. By lemma 13 and IH, we have $\nu([n_1'/x]n_2') \Rightarrow_{\beta\mu} \nu([n_1'''/x]n_2''')$ and $\nu([n_1''/x]n_2'') \Rightarrow_{\beta\mu} \nu([n_1'''/x]n_2''')$.

  The other cases are either similar to the one above or easy.

□

One may also use Takahashi's method [47] to prove the lemma above. We will not explore that here.

Lemma 15. *$\nu(\vec{\mu}\vec{\mu}t) \equiv \nu(\vec{\mu}t)$ and $\nu(\vec{\mu}([t_2/x]t_1)) \equiv \nu([\vec{\mu}t_2/x]\vec{\mu}t_1)$*

Lemma 16. *If $a \to_\beta b$, then $\nu(a) \to_{\beta\mu} \nu(b)$.*

Proof. We prove this by induction on the derivation(depth) of $a \to_\beta b$. We list a few non-trial cases:

- **Base Case:** $\dfrac{(x_i \mapsto t_i) \in \mu}{\mu x_i \to_\beta \mu t_i}$

  We have $\nu(\mu x_i) \equiv \mu x_i \to_{\beta\mu} \nu(\mu t_i)$.

- **Base Case:** $\overline{(\lambda x.t)t' \to_\beta [t'/x]t}$

  We have $\nu((\lambda x.t)t') \equiv (\lambda x.\nu(t))\nu(t') \to_{\beta\mu} \nu([\nu(t)/x]\nu(t')) \equiv \nu([t'/x]t)$.

- **Step Case:** $\dfrac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'}$

  By IH, we have $\nu(\lambda x.t) \equiv \lambda x.\nu(t) \overset{IH}{\to_{\beta\mu}} \lambda x.\nu(t') \equiv \nu(\lambda x.t')$.

- **Step Case:** $\dfrac{t \to_\beta t'}{\mu t \to_\beta \mu t'}$

  We want to show $\nu(\mu t) \to_{\beta\mu} \nu(\mu t')$. If $\mathrm{dom}(\mu)\#\mathrm{FV}(t)$, then $\nu(\mu t) \equiv \nu(t) \overset{IH}{\to_{\beta\mu}} \nu(t') \equiv \nu(\mu t')$. Of course, here we assume beta-reduction does not introduce any new variable.

  If $\mathrm{dom}(\mu) \cap \mathrm{FV}(t) \neq \emptyset$, then identify t as $\dot{\overrightarrow{\mu_1}}t''$, where t'' does not contain any closure at head position. We do case analyze on the structure of t'':

  - **Case.** $t'' = x_i \in \mathrm{dom}(\dot{\overrightarrow{\mu_1}})$ or $x_i \notin \mathrm{dom}(\dot{\overrightarrow{\mu_1}})$, these cases will not arise.

  - **Case.** $t'' = \lambda y.t_1$, then it must be that $t' = \dot{\overrightarrow{\mu_1}}(\lambda y.t_1')$ where $t_1 \to_\beta t_1'$. So we get $\mu\dot{\overrightarrow{\mu_1}}t_1 \to_\beta \mu\dot{\overrightarrow{\mu_1}}t_1'$. By IH(depth of $\mu\dot{\overrightarrow{\mu_1}}t_1 \to_\beta \mu\dot{\overrightarrow{\mu_1}}t_1'$ is smaller), we have $\nu(\mu\dot{\overrightarrow{\mu_1}}t_1) \to_{\beta\mu} \nu(\mu\dot{\overrightarrow{\mu_1}}t_1')$. Thus $\nu(\mu\dot{\overrightarrow{\mu_1}}(\lambda y.t_1)) \equiv \lambda y.\nu(\mu\dot{\overrightarrow{\mu_1}}t_1) \to_{\beta\mu} \lambda y.\nu(\mu\dot{\overrightarrow{\mu_1}}t_1') \equiv \nu(\mu\dot{\overrightarrow{\mu_1}}(\lambda y.t_1'))$.

  - **Case.** $t'' = t_1 t_2$ and $t' = \dot{\overrightarrow{\mu_1}}(t_1' t_2)$, where $t_1 \to_\beta t_1'$. We have $\mu\dot{\overrightarrow{\mu_1}}t_1 \to_\beta \mu\dot{\overrightarrow{\mu_1}}t_1'$. By IH(depth of $\mu\dot{\overrightarrow{\mu_1}}t_1 \to_\beta \mu\dot{\overrightarrow{\mu_1}}t_1'$ is smaller), $\nu(\mu\dot{\overrightarrow{\mu_1}}t_1) \to_{\beta\mu} \nu(\mu\dot{\overrightarrow{\mu_1}}t_1')$. Thus $\nu(\mu\dot{\overrightarrow{\mu_1}}(t_1 t_2)) \equiv \nu(\mu\dot{\overrightarrow{\mu_1}}t_1)\nu(\mu\dot{\overrightarrow{\mu_1}}t_2) \to_{\beta\mu} \nu(\mu\dot{\overrightarrow{\mu_1}}t_1')\nu(\mu\dot{\overrightarrow{\mu_1}}t_2) \equiv \nu(\mu\dot{\overrightarrow{\mu_1}}(t_1' t_2))$. For $t'' = t_1 t_2'$, where $t_2 \to_\beta t_2'$, we can argue similarly.

  - **Case.** $t'' = (\lambda y.t_1)t_2$ and $t' = \dot{\overrightarrow{\mu_1}}([t_2/y]t_1)$. We have $\nu(\mu\dot{\overrightarrow{\mu_1}}((\lambda y.t_1)t_2)) \equiv$

$(\lambda y.\nu(\mu\overrightarrow{\mu_1}t_1)))\nu(\mu\dot{\overrightarrow{\mu_1}}t_2) \rightarrow_{\beta\mu} \nu([\nu(\mu\dot{\overrightarrow{\mu_1}}t_2)/y]\nu(\mu\overrightarrow{\mu_1}t_1)) \equiv \nu([\mu\dot{\overrightarrow{\mu_1}}t_2/y]\mu\overrightarrow{\mu_1}t_1) \equiv$ $\nu(\mu\dot{\overrightarrow{\mu_1}}[t_2/y]t_1)$(lemma 15).

□

Theorem 2. $\rightarrow_\beta \cup \rightarrow_\mu$ *is confluent.*

Proof. We know by diamond property of $\Rightarrow_{\beta\mu}$, $\rightarrow_{\beta\mu}$ is confluent. Since $\rightarrow_\mu$ is strongly normalizing and confluent, and by lemma 16 and Hardin's interpretation lemma(lemma 7), we conclude $\rightarrow_\beta \cup \rightarrow_\mu$ is confluent. □

CHAPTER 4

AN ATTEMPT TO EXPRESSIVE TYPE THEORY THROUGH INTERNALIZATION

This Chapter introduces the concept of *internalization structure*, which can be used to incorporate certain relations into $\mathbf{F}^{\Pi}$, a variant of system **F**, while maintaining termination of the new system. We will call this process of incorporation *internalization*, $\mathbf{F}^{\Pi}$ the *base system* and the new system after the incorporation the *internalized system*. We first specify the syntax, and then the semantics of $\mathbf{F}^{\Pi}$ via the Tait-Girard reducibility method (Section 4.2). We then define internalization structure (Section 4.3). We show that we can obtain a terminating internalized system from an internalization structure (Section 4.4). As motivating examples, we demonstrate how our framework can be applied to internalize subtyping, full-beta term equality and term-type inhabitation relations (Section 4.5). Finally, we discuss some of the difficulties in Section 4.6.

4.1 Backgrounds

Type systems often incorporate auxiliary judgments in their typing relations. For example, the subsumption rule for subtyping:

$$\frac{\Gamma \vdash t : T \quad T <: T'}{\Gamma \vdash t : T'}\ sub$$

Likewise, the conversion rule for type-equivalence:

$$\frac{\Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'}\ conv$$

We propose a framework for incorporating the meta-level relations such as $<:$ and $\equiv$ as types in the type system, and shows that such extensions yield terminationing systems under the call-by-name reduction. We call the deduction systems producing auxiliary judgments *metasystems*, and refer to the typing rules that modify types based such metasystem judgments as *automatic conversion rules*. We will also consider cut-down type systems without automatic conversion rules, which are called *base systems*. For instance, the subtyping and the type-equivalence derivation systems are the metasystems, and rules *sub* and *conv* are the automatic conversion rules. We will discuss an variant of System **F**, which is the base system for our extensions.

Type structure can be used to reflect metasystem judgments. Indeed this has been done in several languages: equality sets in Martin-Löf type theory enable reasoning about equality relations [38]; Sjöberg and Stump's T^{vec} uses types to reflect call-by-value term equality in the presence of divergence [46], and the AuraConf language uses proofs of type e **isa** t to indicate expression e may be cast to type t [50].

4.2 The Base system $\mathbf{F}^{\Pi}$

Internalization builds off of base system $\mathbf{F}^{\Pi}$, a variant of system **F**.

Definition 27 (Syntax and Reductions)**.**

Types $T \ ::= \ B \mid X \mid \Pi x : T.T \mid \forall X.T$

Terms $t, u \ ::= \mathsf{axiom} \mid x \mid (t\ t) \mid \lambda x.t$

Contexts $\mathcal{C}$ ::= $\cdot \mid \mathcal{C}\ t$

Values v ::= $\lambda x.t \mid \mathsf{axiom}$

Reductions $\mathcal{C}[(\lambda x.t)\ t'] \rightsquigarrow \mathcal{C}[[t'/x]t]$

Note that we use call-by-name reduction strategy.

Definition 28 (Kinding). $\boxed{\Gamma \vdash \mathsf{OK}}$

$$\frac{}{\cdot \vdash \mathsf{OK}} \qquad \frac{\Gamma \vdash \mathsf{OK}}{\Gamma, X \vdash \mathsf{OK}} \qquad \frac{\Gamma \vdash \mathsf{OK} \quad \mathrm{FVar}(T) \subseteq \mathrm{dom}(\Gamma)}{\Gamma, x : T \vdash \mathsf{OK}}$$

FVar(T) means the set of free type variables and free term variables in type T. dom(Γ) means the domain of the context, i.e., $e \in \mathrm{dom}(\Gamma)$ iff e is either a type variable such that $\Gamma \equiv \Gamma_1, e, \Gamma_2$, or a term variable such that $\Gamma \equiv \Gamma_1, e : T, \Gamma_2$.

Definition 29 (Typing). $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma, x : T_1 \ \vdash t : T_2}{\Gamma \vdash \lambda x.t : \Pi x : T_1.T_2}\ \Pi\text{-}\mathit{intro} \qquad \frac{(x : T) \in \Gamma \quad \Gamma \vdash \mathsf{OK}}{\Gamma \vdash x : T}\ \mathit{Var}$$

$$\frac{\Gamma \vdash t_1 : \Pi x : T_1.T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : [t_2/x]T_2}\ \Pi\text{-}\mathit{elim} \qquad \frac{\Gamma, X \vdash t : T}{\Gamma \vdash t : \forall X.T}\ \forall\text{-}\mathit{intro}$$

$$\frac{\Gamma \vdash t : \forall X.T \quad \mathrm{FVar}(T') \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash t : [T'/X]T}\ \forall\text{-}\mathit{elim}$$

The differences between $\mathbf{F}$ and $\mathbf{F}^{\Pi}$ are as follows:

1. $\mathbf{F}^{\Pi}$ is parametrized by a finite set B of constant types and it contains constant terms like axiom. Later, we will use axiom to inhabit special types.

2. The notion of value is extended by including constant terms.

3. $\mathbf{F}^{\Pi}$ uses dependent product Π instead of arrow $\to$ as the function type constructor anticipating the use of types that mention terms.

A word about the use of call-by-name reduction is warranted. The main result of this paper is normalization for systems derived by internalization from the base system $\mathbf{F}^{\Pi}$. Strong normalization does not hold for all such systems, as we show by example in Section 4.5.1. So (weak) normalization is all that we can obtain. An interesting result of our investigation into internalization is that normalization with respect to call-by-name reduction imposes fewer requirements on internalization structures than with call-by-value reduction. Specifically, the λ-abstraction case of the proof of Theorem 3 goes through more directly using call-by-name reduction; with call-by-value reduction, dependent typing imposes additional restrictions.

4.2.1 Interpretation of Types in $\mathbf{F}^{\Pi}$

Reducibility is a well-known technique for proving the normalization of type systems such as $\mathbf{F}$. In this paper, we use it to interpret $\mathbf{F}^{\Pi}$'s types. Reducibility will both provide intuition for $\mathbf{F}^{\Pi}$'s semantics and yield a normalization result.

Definition 30. *A reducibility candidate $\mathcal{R}$ is a set of terms that satisfies the following conditions:*

CR 1 *If $t \in \mathcal{R}$,then $t \in \mathcal{V}$, where $\mathcal{V}$ is the set of closed terms that that reduces to a value.*

CR 2 *If $t \in \mathcal{R}$ and $t \rightsquigarrow t'$, then $t' \in \mathcal{R}$.*

CR 3 *If t is a closed term, $t \rightsquigarrow t'$ and $t' \in \mathcal{R}$, then $t \in \mathcal{R}$.*

Definition 31. *Let $\mathfrak{R}$ be the set of all reducibility candidates. Let $TVar$ be the set of all type variables. Let ϕ be a finite function with $\mathrm{dom}(\phi) \subseteq TVar$ and $\mathrm{range}(\phi) \subseteq \mathfrak{R}$. If $\mathrm{dom}(\phi) = \{X_1, X_2, ...X_n\}$, then we usually write ϕ as $[\mathcal{R}_1/X_1, ...\mathcal{R}_n/X_n]$.*

Definition 32 (Interpretation of Types)**.**

$t \in [\![B]\!]_\phi$ iff $t \in \mathcal{R}_B$, where $\mathcal{R}_B \in \mathfrak{R}$.

$t \in [\![X]\!]_\phi$ iff $t \in \phi(X)$.

$t \in [\![\Pi x : T_1.T_2]\!]_\phi$ iff $t \in \mathcal{V}$ and $(\forall u \in [\![T_1]\!]_\phi \Rightarrow (t\ u) \in [\![[u/x]T_2]\!]_\phi)$.

$t \in [\![\forall X.T]\!]_\phi$ iff $\forall \mathcal{R} \in \mathfrak{R}, t \in [\![T]\!]_{\phi[\mathcal{R}/X]}$.

Note that constant types B and their interpretations $\mathcal{R}_B$ are left unspecified; these may be filled in later. For any $[\![T]\!]_\phi$, let $\mathrm{FV}(T)$ be the set of free type variable in T.we assume $\mathrm{FV}(T) \subseteq \mathrm{dom}(\phi)$.

4.2.2 Type Soundness

The theorem below shows that any typable closed term is normalizing, and can be shown in a standard way using Tait-Girard reducibility (cf. [24]). Several properties of the interpretation of types are required, which can all be proved by induction on the structure of types in $\mathbf{F}^\Pi$.

Lemma 17. *$[\![T]\!]_\phi \in \mathfrak{R}$, in the other words, the interpretation of a type is indeed a reducibility candidate.*

Definition 33. *We define the set $[\Gamma]$ of well-typed substitutions (σ, δ) w.r.t. Γ as follows:* $\dfrac{}{(\emptyset, \emptyset) \in [.]}$ $\dfrac{(\sigma, \delta) \in [\Gamma] \quad \mathcal{R} \in \mathfrak{R}}{(\sigma, \delta \cup \{(X, \mathcal{R})\}) \in [\Gamma, X]}$ $\dfrac{(\sigma, \delta) \in [\Gamma] \quad t \in [\![\sigma T]\!]_\delta}{(\sigma \cup \{(x, t)\}, \delta) \in [\Gamma, x : T]}$

Theorem 3 (Type Soundness)**.** *If $\Gamma \vdash t : T$, then $\forall (\sigma, \delta) \in [\Gamma], (\sigma\ t) \in [\![\sigma T]\!]_\delta$.*

4.3 Internalized Structure

An internalization structure is a triple $(D, E, \mathcal{I})$. *Reflective relational sentences* D define the syntax of metasystem propositions and identify valid metasystem judgments. *Elimination relation* E defines automatic conversion rules based on judgments from D. Finally, *interpretation* $\mathcal{I}$ defines semantics for reflective relational sentences as relations over the sets of terms in the base system. All internalization structures require that D and E are *sound*. As a central result of our work, we show that any sound internalized system constructed from an internalization structure is guaranteed to be terminating. Internalization is based on internalization structure. The internalization structure contains the information of how to construct reflective relational sentences and how these reflective relational sentences interact with the base system. It also gives the meaning of the reflective relational sentences through the interpretation of types in the base system. Once we obtain a sound internalization structure, we can then begin the process of internalization by first incorporating the reflective relational sentences as types, then add two new typing rules to deal with these reflective relational sentences.

4.3.1 Reflective Relational Sentence-D

We define the kind of judgments or relations that could be integrated into the base system. Essentially these are the relations on the terms and types from the base system.

Definition 34. *Let signature* $\Sigma \subseteq \mathbf{Symbols} \times \mathbb{N} \times \mathbb{N}$*, where* **Symbols** *means a set of relation symbols, and* $\mathbb{N}$ *is the set of natural numbers.* $R^{n\times m} \in \Sigma$ *means* $R \in \mathbf{Symbols}$ *and the arity of* R *is* $n+m$.

Definition 35. *A relational sentence on the basic system is a syntactic object of form* $R^{(n\times m)}(t_1, ..., t_n, T_1, ..., T_m)$*, where* t, T *are defined in* $\boldsymbol{F}^{\Pi}$ *and* $R^{(n\times m)} \in \Sigma$.

Definition 36. *Let $\mathfrak{A}$ be the set of all relational sentences. A set of reflective relational sentences D is a subset of all relational sentences, i.e. $D \subseteq \mathfrak{A}$.*

Reflective relational sentences are used to formalize a metasystem's derivable judgments. When we define specifically how to recognize the reflective relational sentences from relational sentences, we obtain a kind of metasystem. This metasystem need not be recursive.

4.3.2 Elimination Relation-E

An elimination relation is a syntactic constraint used to specify how the metasystem influences the base system. We will appeal to an elimination relation when we add the elimination rule to the base system for the reflective relational sentences. Since the elimination relation is used after internalizing reflective relational sentences as types, we need to extend the definition of types and the context accordingly.

Definition 37. *We define extended types and extended contexts as follows:*

RTypes $A \ ::= \ R_1^{(n\times m)}(t_1, ..., t_n, T_1, ..., T_m) \mid ... \mid R_l^{(n\times m)}(t_1, ..., t_n, T_1, ..., T_m)$

ETypes $S \ ::= A \mid B \mid X \mid \Pi x : S.S \mid \forall X.S$

EContext $\Delta \ ::= \ \cdot \mid \Delta, x : S \mid \Delta, X$

Definition 38. *We specify an elimination relation E by:*

$E \subseteq$ **EContext** $\times$ **Terms** $\times$ **Terms** $\times\, \mathfrak{A} \,\times$ **ETypes** $\times$ **ETypes**.

For example, when we consider the specific internalization structure for subtyping below, we will define an elimination relation where $(\Delta, t, t', T < T', T, T') \in E$ holds iff in extended context Δ, t has the type T, t' has the type $T < T'$, and we can change the type of t to T'.

4.3.3 Interpretation-$\mathcal{I}$

We defined the interpretation of types of $\mathbf{F}^{\Pi}$ before. Since interpretation of types is a set of terms and the reflective relational sentences are relations about between terms and types in $\mathbf{F}^{\Pi}$, it is natural to understand the meaning of these reflective relational sentences as set-theoretic relations between interpretation of types. Take subtyping as an example; we interpret subtype judgment $<:$ as the subset relation $\subseteq$ on interpretation of types[1]. Interpretation-$\mathcal{I}$ is defined to capture this intuition. Later we will relate interpretation-$\mathcal{I}$ to reflective relational sentences and elimination relation through two soundness properties.

Definition 39. *Let $\mathfrak{R}$ be the set of all reducibility candidates as defined in $\boldsymbol{F}^{\Pi}$. We define an interpretation of $R^{(n\times m)}$–$\mathcal{I}_{R^{(n\times m)}}$ to be $\mathcal{I}_{R^{(n\times m)}} \subseteq \mathbf{Terms}^n \times \mathfrak{R}^m$.*

4.3.4 Soundness Properties

Now that we have defined all parts of an internalization structure, we can formulate two soundness properties for an internalization structure. Since one of the soundness properties is related to the extended types, we first define the interpretation for extended types. Then we identify the soundness properties.

Definition 40. *Let ϕ be an environment function w.r.t. type S, which is defined in the same way as definition 31 except we extend it to type S. Let $\mathcal{A}$ be the set of closed terms that normalize at* axiom*. The interpretation of types $[\![S]\!]_{\phi}$ is defined inductively*

[1]This is also observed by [44]

as follows:

- $t \in [\![B]\!]_\phi$ *iff* $t \in \mathcal{R}_B$.
- $t \in [\![R^{(n\times m)}(t_1, ..., t_n, T_1, ..., T_m)]\!]_\phi$ *iff* $t \in \mathcal{A}$ *and* $(t_1, ..., t_n, [\![T_1]\!]_\phi, ..., [\![T_m]\!]_\phi) \in \mathcal{I}_{R^{(n\times m)}}$.
- $t \in [\![X]\!]_\phi$ *iff* $t \in \phi(X)$.
- $t \in [\![\Pi x : S_1.S_2]\!]_\phi$ *iff* $t \in \mathcal{V}$ *and* $(\forall u \in [\![S_1]\!]_\phi \Rightarrow (t\ u) \in [\![[u/x]S_2]\!]_\phi)$.
- $t \in [\![\forall X.S]\!]_\phi$ *iff* $\forall \mathcal{R} \in \mathfrak{R}, t \in [\![S]\!]_{\phi[\mathcal{R}/X]}$.

We define $(\sigma, \delta) \in [\Delta]$ in the same way as $(\sigma, \delta) \in [\Gamma]$, except with extended contexts and extended types.

Definition 41. *We say a tuple $\langle D, E, \mathcal{I}\rangle$ is an internalization structure iff it satisfies the following soundness properties:*

Soundness of reflective relational sentences:

If $R^{(n\times m)}(t_1, ..., t_n, T_1, ..., T_m) \in D$, then $\forall \phi, \sigma, (\sigma t_1, ..., \sigma t_n, [\![\sigma T_1]\!]_\phi, ..., [\![\sigma T_m]\!]_\phi) \in \mathcal{I}_{R^{(n\times m)}}$.

Soundness of the elimination relation:

Suppose $(\Delta, t, t', R^{(n\times m)}(t_1, ..., t_n, T_1, ..., T_m), S, S') \in E$, $(\sigma, \delta) \in [\Delta]$, $\sigma(t) \in [\![\sigma S]\!]_\delta$ and $R^{(n\times m)}(t_1, ..., t_n, T_1, ..., T_m) \in D$. Then $\sigma(t) \in [\![\sigma S']\!]_\delta$.

Soundness of reflective relational sentences means that the reflective relational sentences are a conservative approximation of interpretation-$\mathcal{I}$. *Soundness of the elimination relation* will imply that the elimination rule for internalized systems respects the Girard-Tait type interpretation and is semantically compatible with substitutions that arise duing CBN evaluation.

4.4 Internalized System

We have defined the internalization structure–$(D, E, \mathcal{I})$. Using an internalization structure, we can construct a new system–we call it the internalized system–from the internalization structure and $\mathbf{F}^{\Pi}$. The term syntax and operational semantics of internalized system are the same as $\mathbf{F}^{\Pi}$, while the syntax of types and contexts are the $\mathbf{RTypes}, \mathbf{ETypes}, \mathbf{EContexts}$ in definition 37. The well-formed extended context $\Delta \vdash \mathbf{OK}$ is defined just as before except using **EContexts**.

Definition 42. $\boxed{\Delta \vdash t : S}$

$$\frac{A \in D \quad \mathrm{FVar}(A) \subseteq \mathrm{dom}(\Delta) \quad \Delta \vdash \mathsf{OK}}{\Delta \vdash \mathsf{axiom} : A}\ A\textit{-intro}$$

$$\frac{\Delta(x) = S \quad \Delta \vdash \mathsf{OK}}{\Delta \vdash x : S}\ \textit{Var}$$

$$\frac{\Delta \vdash t : T \quad \Delta \vdash t' : A \quad E(\Delta, t, t', A, T, T')}{\Delta \vdash t : T'}\ A\textit{-elim}$$

$$\frac{\Delta, x : S_1 \ \vdash t : S_2}{\Delta \vdash \lambda x.t : \Pi x : S_1.S_2}\ \Pi\_\textit{intro}$$

$$\frac{\Delta \vdash t_1 : \Pi x : S_1.S_2 \quad \Delta \vdash t_2 : S_1}{\Delta \vdash t_1\ t_2 : [t_2/x]S_2}\ \Pi\_\textit{elim}$$

$$\frac{\Delta, X \vdash t : S}{\Delta \vdash t : \forall X.S}\ \forall\_\textit{intro}$$

$$\frac{\Delta \vdash t : \forall X.S \quad [S'/X]S \in \mathbf{ETypes} \quad \mathrm{FVar}(S') \subseteq \mathrm{dom}(\Delta)}{\Delta \vdash t : [S'/X]S}\ \forall\_\textit{elim}$$

We can see that the new type assignment system contains two new rules:A-

intro and A-elim. The A-intro rule is used to introduce reflective relational sentences as types in the internalized system, while the A-elim rule is for using the reflective relational sentences to change the type of a term accordingly. The theorem below guarantees that the internalized system generated from $\mathbf{F}^{\Pi}$ and internalization structure is *terminating*, which is the central result of internalization.

Theorem 4 (Type Soundness)**.** *If* $(D, E, \mathcal{I})$ *is an internalization structure and* $\Delta \vdash t : S$*, then* $\forall(\sigma, \delta) \in [\Delta], (\sigma\ t) \in [\![\sigma S]\!]_{\delta}$*.*

Corollary 3. *If* $\cdot \vdash t : S$*, then* $t \in \mathcal{V}$*.*

Because typing contexts may introduce spurious assumptions, some open contexts may assign a type to a diverging term. Section 4.5.1 shows gives an example. This is an expected outcome of reasoning from invalid premises. Indeed Corollary 3 may be strengthened to allow contexts where all variables are classified by inhabited types.

4.5 Examples

In previous section, we capsule our development of internalized system as constructing a sound internalization structure. Now let us see how we can apply our formalization of internalization to internalize subtyping, full-beta term equality and term-type inhabitation relations as types. First, we specify an instance of $\mathbf{F}^{\Pi}$. Namely, we instantiate constant types as $B ::= \top \mid \bot$. Additionally, we define $[\![\bot]\!]_{\phi} := \emptyset, [\![\top]\!]_{\phi} := \mathcal{V}$.

Recall that internalization works as follows. We first define the set of reflective relational sentences that contains all the derivable judgments from subtyping, full-beta term equality and term-type inhabitation. Then we define the elimination relation and interpretation. We show our definition interpretation structure is sound. Finally we present the internalized system as the result of internalization. We will follow this recipe in the sequel.

4.5.1 Subtyping

We need to instantiate the three parts of internalization structure-$\langle D, E, I \rangle$. First, we specify $\Sigma := \{<^{0+2}\}$. Then we know all the reflective relational sentences should be in the form $T_1 < T_2$. We identify reflective relational sentences D as follows:

Definition 43. $\boxed{T < T' \in D}$

$$\frac{}{T < \top \in D} \qquad \frac{}{\bot < T \in D} \qquad \frac{}{X < X \in D}$$

$$\frac{T_1 < T_2 \in D}{\forall X.T_1 < \forall X.T_2 \in D} \qquad \frac{T_1' < T_1 \in D \quad T_2 < T_2' \in D}{\Pi x : T_1.T_2 < \Pi x : T_1'.T_2' \in D}$$

We can see that the way we identify D is similar to the way we write subtyping rules. Now we define $(\Delta, t, t', T < T', T, T') \in E$. The meaning of this elimination relation is that if t has type T in context Δ and t' has type $T < T'$, then t can also has the type T'. We define: $\mathcal{I}_< := \{(\mathcal{R}_1, \mathcal{R}_2) \mid \mathcal{R}_1 \subseteq \mathcal{R}_2\}$. We can see that $\mathcal{I}_<$ capture all the subset relations on reducibility candidates. The following two lemmas make sure we obtain a sound internalization structure from $(D, E, \mathcal{I}_<)$ we defined above.

Lemma 18 (Soundness of the Reflective Relational Sentence). *If* $(T < T') \in D$*, then* $\forall \sigma, \forall \phi, ([\![\sigma T]\!]_\phi, [\![\sigma T']\!]_\phi) \in \mathcal{I}_<$.

Proof. Since $[\![\sigma T]\!]_\phi = [\![T]\!]_\phi$, we just need to show: If $(T < T') \in D$, then $\forall \phi, ([\![T]\!]_\phi, [\![T']\!]_\phi) \in \mathcal{I}_<$. We will prove this by induction on the structure of T.

- **Case**: $T = \top$ or $T = \bot$

  By inversion, it holds.

- **Case**: $T = X$

  By inversion, we know $T' = X$ *or* $\top$, again, it is the case.

- **Case**: $T = \Pi x : T_1.T_2$

  By inversion, $T' = \top$ or $T' = \Pi x : T_1'.T_2'$. Let us consider $T' = \Pi x : T_1'.T_2'$. In this case, by inversion, $T_1' < T_1 \in D, T_2 < T_2' \in D$. By IH, we have $[\![T_1']\!]_\phi \subseteq [\![T_1]\!]_\phi$. Again, by IH, we have $[\![T_2]\!]_\phi \subseteq [\![T_2']\!]_\phi$. For any $u \in [\![T_1']\!]_\phi \subseteq [\![T_1]\!]_\phi$, if $t \in [\![\Pi x : T_1.T_2]\!]_\phi$, we have $tu \in [\![[u/x]T_2]\!]_\phi = [\![T_2]\!]_\phi \subseteq [\![T_2']\!]_\phi$. So $t \in [\![\Pi x : T_1'.T_2']\!]_\phi$.

- **Case**: $T = \forall X.T$

  By inversion, $T' = \top$ or $\forall X.T'$. So let's consider $T' = \forall X.T'$. By inversion, we know $T < T' \in D$. We know for $t \in [\![\forall X.T]\!]_\phi$, $\forall \mathcal{R} \in \mathfrak{R}, t \in [\![T]\!]_{\phi[\mathcal{R}/X]}$. By IH, $[\![T]\!]_{\phi[\mathcal{R}/X]} \subseteq [\![T']\!]_{\phi[\mathcal{R}/X]}$. So $t \in [\![\forall X.T']\!]_\phi$.

$\square$

Lemma 19 (Soundness of the Elimination Relation)**.** *If* $(\Delta, t, t', T_1 < T_2, T_1, T_2) \in E$, $(\sigma, \delta) \in [\Delta]$ *and* $\sigma(t) \in [\![\sigma T_1]\!]_\delta = [\![T_1]\!]_\delta$ *and* $T_1 < T_2 \in D$*, then* $\sigma(t) \in [\![\sigma T_2]\!]_\delta = [\![T_2]\!]_\delta$*.*

The subtyping setting also provides an example that it is possible to have diverging term under open terms and full-beta reduction in internalized system. It is possible to derive $y : (\top < (\top \to \top)) \vdash (\lambda x.xx)(\lambda x.xx) : \top$ using the underivable fact $\top < (\top \to \top)$ and derivable $(\top \to \top) < \top$ to establish an isomorphism between types $\top$ and $\top \to \top$. Sticking to closed terms means we need not worry about this derivation directly. And call-by-name evaluation ensures that $\cdot \vdash \lambda y.(\lambda x.xx)(\lambda x.xx) : (\top < (\top \to \top)) \to \top$ does not reduce. In contrast, full reduction would loop.

4.5.2 Term Equality and Term-Type Inhabitation

We can go even further to explore the internalization structure. We add two more relation symbols to signature so that $\Sigma = \{\downarrow^{(2+0)}, <^{(0+2)}, \lhd^{(1+1)}\}$. For simplicity, we usually do not specify the arity. Thus the relational sentences have form: $t_1 \downarrow t_2, T_1 < T_2$, and $t \lhd T$ for base-system t and T.

Now we are ready to specify more reflective relational sentences. We define $\lhd$ reflective relational sentences by the following condition:

$$t \lhd T \in D \text{ iff } \forall \phi, t \in [\![T]\!]_\phi$$

Notice that this definition is not algorithmic, which is fine since our framework does not require decidability for the set D for reflective relational sentences.

The $\lhd$ symbol allows us to give "morally correct" types to terms which cannot otherwise be checked. In practice, such terms are created when extracting computational content from mechanically checked proofs. As a concrete example, the Coq proof assistant uses an expressive language to define functional programs and exports that code to OCaml for efficient compilation. Resulting OCaml programs do not go wrong, but must use `Obj.magic:`$\alpha \to \beta$ to pass ML's weaker type system. Likewise, AuraConf [50] uses a type constructor resembling $\triangleleft$ to inform the type checker about the concealed types of opaque ciphertexts. Note that weaker variants of $\triangleleft$ may be possible when, as in the case of extracted proofs, there is a conservative procedure for checking semantic type inclusion, $t \triangleleft_{alt} T \in D$ iff $\mathit{Oracle}(t, T)$. (We do not consider such variants further.)

We define $t_1 \downarrow t_2 \in D$ by the following rules:

Definition 44. $\boxed{t \downarrow t' \in D}$

$$\frac{}{t \downarrow t \in D} \qquad \frac{}{(\lambda x.t)t' \downarrow [t'/x]t \in D} \qquad \frac{t_1 \downarrow t_2 \in D}{t_1\ t \downarrow t_2\ t \in D}$$

$$\frac{t_1 \downarrow t_2 \in D}{\lambda x.t_1 \downarrow \lambda x.t_2 \in D} \qquad \frac{t_1 \downarrow t_2 \in D}{t\ t_1 \downarrow t\ t_2 \in D} \qquad \frac{t_1 \downarrow t_2 \in D \quad t_2 \downarrow t_3 \in D}{t_1 \downarrow t_3 \in D}$$

$$\frac{t_1 \downarrow t_2 \in D}{t_2 \downarrow t_1 \in D}$$

The rules above are the same as how we define the conversion in lambda calculus. In this case, the syntax of extended types (as defined by the internalization framework) is:

$$\textbf{EType}\ S ::= \top \mid \bot \mid X \mid \Pi x : S.S \mid \forall X.S \mid t_1 \downarrow t_2 \mid T_1 < T_2 \mid t \triangleleft T$$

The additional elimination relations are:

$$(\Delta, t, t', t_1 \downarrow t_2, [t_1/x](t_3 \downarrow t_4), [t_2/x](t_3 \downarrow t_4)) \in E.$$

$$(\Delta, t, t', t \triangleleft T', T, T') \in E$$

The additional interpretations $\mathcal{I}_{\downarrow}, \mathcal{I}_{\triangleleft}$ are:

- $\mathcal{I}_{\downarrow} \subseteq$ **Terms** $\times$ **Terms** defined by $\mathcal{I}_{\downarrow} := \{(t_1, t_2) \mid t_1 \downarrow t_2 \in D\}$.

- $\mathcal{I}_\triangleleft \subseteq \textbf{Terms} \times \mathfrak{R}$ defined by $\mathcal{I}_\triangleleft := \{(t, \mathcal{R}) \mid t \in \mathcal{R}\}$.

We have now defined the three parts of the internalization structure. We need to show that this structure is sound. For that purpose, we have following lemmas.

Lemma 20 (Soundness of the Reflective Relational Sentence)**.**

- *If* $(t_1 \downarrow t_2) \in D$*, then* $\forall \sigma, (\sigma t_1, \sigma t_2) \in \mathcal{I}_\downarrow$.

- *If* $(t \triangleleft T) \in D$*, then* $\forall \sigma, \forall \phi, (\sigma t, [\![\sigma T]\!]_\phi) \in \mathcal{I}_\triangleleft$.

Proof. If $(t_1 \downarrow t_2) \in D$, we have $\forall \sigma, (\sigma t_1, \sigma t_2) \in D$. This is because we define the $t \downarrow t'$ relation same as the conversion in lambda calculus and this is one of its properties. Thus $(\sigma t_1, \sigma t_2) \in \mathcal{I}_\downarrow$ by definition of $\mathcal{I}_\downarrow$.

If $(t \triangleleft T) \in D$, by definition, we have $\forall \phi, t \in [\![T]\!]_\phi$. Since t is closed, $\forall \sigma, \sigma t \equiv t$. And we have $[\![\sigma T]\!]_\phi = [\![T]\!]_\phi$. So $\forall \phi, \forall \sigma, \sigma t \in [\![\sigma T]\!]_\phi$. Thus $\forall \sigma, \forall \phi, (\sigma t, [\![\sigma T]\!]_\phi) \in \mathcal{I}_\triangleleft$.

$\square$

Lemma 21 (Soundness of the Elimination Relation)**.**

- *If* $(\Delta, t, t', t_1 \downarrow t_2, [t_1/x](t_3 \downarrow t_4), [t_2/x](t_3 \downarrow t_4)) \in E$*,* $(\sigma, \delta) \in [\Delta]$ *and* $\sigma(t) \in [\![\sigma[t_1/x](t_3 \downarrow t_4)]\!]_\delta$ *and* $t_1 \downarrow t_2 \in D$*, then* $\sigma(t) \in [\![\sigma[t_2/x](t_3 \downarrow t_4)]\!]_\delta$.

- *If* $(\Delta, t, t', t \triangleleft T', T, T') \in E$*,* $(\sigma, \delta) \in [\Delta]$ *and* $\sigma(t) \in [\![\sigma T]\!]_\delta$ *and* $t \triangleleft T' \in D$*, then* $\sigma(t) \in [\![\sigma T']\!]_\delta$.

Proof. We have $\sigma(t) \in [\![\sigma[t_1/x](t_3 \downarrow t_4)]\!]_\delta$, thus $\sigma(t) \in \mathcal{A}$ and $(\sigma[t_1/x]t_3) \downarrow (\sigma[t_1/x]t_4) \in D$. Since $t_1 \downarrow t_2 \in D$, then we have $(\sigma[t_2/x]t_3) \downarrow (\sigma[t_2/x]t_4) \in D$. This is also followed by the property of $t \downarrow t'$. So $\sigma(t) \in [\![\sigma[t_2/x](t_3 \downarrow t_4)]\!]_\delta$.

By *soundness of reflective relational sentences*, $t \lhd T' \in D$ implies $\forall \phi, \sigma t = t \in [\![T']\!]_\phi$. So it is the case.

$\square$

So the structure $(D, E, \mathcal{I}_<, \mathcal{I}_\downarrow, \mathcal{I}_\lhd)$ we have defined is a sound internalization structure. Let us see some instances of *A-elim* rule and *A-intro* rule for the internalized system based on this internalization structure:

$$\frac{t_1 \downarrow t_2 \in D \quad \mathrm{FVar}(t_1 \downarrow t_2) \subseteq \mathrm{dom}(\Delta) \quad \Delta \vdash \mathsf{OK}}{\Delta \vdash \mathsf{axiom} : t_1 \downarrow t_2} \textit{ A-intro}$$

$$\frac{\Delta \vdash t : [t_1/x](t_3 \downarrow t_4) \quad \Delta \vdash t' : t_1 \downarrow t_2}{\Delta \vdash t : [t_2/x](t_3 \downarrow t_4)} \textit{ A-elim}$$

$$\frac{t \lhd T' \in D \quad \mathrm{FVar}(t \lhd T') \subseteq \mathrm{dom}(\Delta) \quad \Delta \vdash \mathsf{OK}}{\Delta \vdash \mathsf{axiom} : t \lhd T'} \textit{ A-intro}$$

$$\frac{\Delta \vdash t : T \quad \Delta \vdash t' : t \lhd T'}{\Delta \vdash t : T'} \textit{ A-elim}$$

We can see that our elimination rule for $\downarrow$ realizes a more general form of transitivity. For example, if we have a term with a type $[t_2/y](t_1 \downarrow y)$ and $t_2 \downarrow t_3 \in D$, then we can assign this term a new type $[t_3/y](t_1 \downarrow y)$ by the elimination rule.

4.6 Summary

We have formalized the notion of internalization structure and demonstrated that the internalized system is terminating. We also have shown how our formalization can be applied to full-beta term equality, subtyping and term-type inhabitation relation. Our approach makes it easier to establish normalization for type theories with these features, since the framework provides the analysis for all but the internalization-specific parts of the language.

In retrospective, the difficulty of this approach is that, with internalization, the type system has the ability to make inconsistent assumption using the internalized relation such as $\triangleleft, <, \downarrow$, which will falsify the type preservation property. For example, assuming we manage to internalize a form of type equivalence that we can automatically convert one type to another, and suppose $a : A \to B \equiv A \to C, d : A \vdash t : A \to B$. Then we can have $a : A \to B \equiv A \to C, d : A \vdash t\ d : B$ and $a : A \to B \equiv A \to C, d : A \vdash t\ d : C$ due to automatic conversion. But we know that B, C are not unifiable. Thus we have a counterexample for type preservation[2]. This counter example will not arise if we adopt the Leibniz equality, namely, we define $T \equiv T'$ as $\forall P.P(T) \to P(T')$. Then we would have $a : A \to B \equiv A \to C, d : A \vdash t\ d : B$ and $a : A \to B \equiv A \to C, d : A \vdash (a\ t)\ d : C$, which is entirely legal and type preservation still hold. This sequence of development suggests that we should at least be careful when we try to "lift" propositional equivalence $\leftrightarrow$ to meta-level equivalence $\equiv$. We will discuss Leibniz equality more in Chapter 6.

[2]This counterexample is found by the Upenn group.

CHAPTER 5

LAMBDA ENCODINGS WITH DEPENDENT TYPES

In this Chapter, we revisit lambda encodings of data, proposing new solutions to several old problems, in particular dependent elimination with lambda encodings (Section 5.2). We start with a type-assignment form of the Calculus of Constructions, restricted recursive definitions and Miquel's implicit product. We add a type construct $\iota x.T$, called a *self type*, which allows T to refer to the subject of typing (Section 5.3). We show how the resulting System **S** with this novel form of dependency supports dependent elimination with lambda encodings, including induction principles (Section 5.4). Strong normalization of **S** is established by defining an erasure from **S** to a version of $\mathbf{F}_\omega$ with positive recursive type definitions, which we analyze (Section 5.5). We also prove type preservation for **S**.

5.1 Introduction

Modern type-theoretic tools Coq and Agda extend a typed lambda calculus with a rich notion of primitive datatypes. Both tools build on established foundational concepts, but the interactions of these, particularly with datatypes and recursion, often leads to unexpected problems. For example, it is well-known that type preservation does not hold in Coq, due to the treatment of coinductive types [22]. Arbitrary nesting of coinductive and inductive types is not supported by the current version of Agda, leading to new proposals like co-patterns [2]. And new issues are discovered with disturbing frequency; e.g., an unexpected incompatibility of extensional consequences of Homotopy Type Theory with both Coq and Agda was discovered in December, 2013 [45].

The above issues all are related to the datatype system, which must determine what are the legal inductive/coinductive datatypes, in the presence of indexing, dependency, and generalized induction (allowing functional arguments to constructors). And for formal study of the type theory – either on paper [52], or in a proof assistant [7] – one must formalize the datatype system, which can be daunting, even in very capable hands (cf. Section 2 of [9]).

Fortunately, an alternative to primitive datatypes exists: lambda encodings, like the well-known Church and Scott encodings [12, 16]. Utilizing the core typed lambda calculus for representing data means that no datatype system is needed at all, greatly simplifying the formal theory. We focus here just on inductive types, since in extensions of System **F**, coinductive types can be reduced to inductive ones [20].

Several problems historically prevented lambda encodings from being adopted in practical type theories. Scott encodings are efficient but do not inherently provide a form of iteration or recursion. Church encodings inherently provide iteration, and are typable in System **F**. Due to strong normalization of System **F** [23], they are thus suitable for use in a total (impredicative) type theory, but:

1. The predecessor of n takes $O(n)$ time to compute instead of constant time.

2. We cannot prove $0 \neq 1$ with the usual definition of $\neq$.

3. Induction is not derivable [21].

4. Large eliminations (computing types from data) are not supported.

These issues motivated the development of the Calculus of Inductive Constructions (cf. [51]). Problem (1) is best known but has a surprisingly underappreciated solution: if we accept positive recursive definitions (which preserve normalization), then we can use Parigot numerals, which are like Church numerals but based on recursors not iterators [40]. Normal forms of Parigot numerals are exponential in size, but a reasonable term-graph implementation should be able to keep them linear via sharing. The other three problems have remained unsolved.

In this Chapter, we propose solutions to problems (2) and (3). For problem (2) we propose to change the definition of falsehood from explosion ($\forall X.X$, everything is true) to equational inconsistency ($\forall X.\Pi x : X.\Pi y : X.x =_X y$, everything is equal for any type). We point out that $0 \neq 1$ is derivable with this notion. Our main contribution is for problem (3). We adapt **CC** to support dependent elimination with Church or Parigot encodings, using a novel type construct called *self types*, $\iota x.T$, to express dependency of a type on its subject. This allows deriving induction principles in a total type theory, and we believe it is the missing piece of the puzzle for dependent typing of pure lambda calculus. For problem (4), we suspect it would be hard to extend self type to support large elimination, that would mean we would have to surport impredicative kind polymorphism, which is known to render Girard's paradox.

We summarize the main technical points:

- System **S**, which enables us to encode Church and Parigot data and derive induction principles for these data.

- We prove strong normalization of **S** by erasure to a version of $\mathbf{F}_\omega$ with positive recursive type definitions. We prove strong normalization of this version of $\mathbf{F}_\omega$ by adapting a standard argument.

- Type preservation for **S** is proved by extending Barendregt's method [6] to handle implicit products and making use of a confluence argument.

Detailed arguments omitted here may be found in [18].

5.2 Overview of System S

System **S** extends a type-assignment formulation of the Calculus of Constructions (**CC**) [15]. We allow global recursive definitions in a form we call a *closure*: $\{(x_i : S_i) \mapsto t_i\}_{i \in N} \cup \{(X_i : \kappa_i) \mapsto T_i\}_{i \in M}$ The x_i are term variables which cannot appear in the terms t_i, but can appear in the types T_i. Occurrences in types are used to express dependency, and are crucial for our approach. Erasure to $\mathbf{F}_\omega$ with positive recursive definitions will drop all such occurrences. The X_i are type variables that can appear positively in the T_i or at erased positions (explained later).

The essential new construct is the self type $\iota x.T$. Note that this is different from self typing in the object-oriented (OO) literature, where the central problem has been to allow self-application while still validating natural record-subtyping rules [39, 1]. Typing the self parameter of an object's methods appears different from allowing a type to refer to its subject, though Hickey proposes a type-theoretic encoding of objects based on very dependent function types $\{f \,|\, x : A \to B\}$, where the range

B can depend on both x and values of the function f itself [27]. The self types we propose appear to be simpler.

5.2.1 Induction Principle

Let us take a closer look at the difficulties of deriving an induction principle for Church numerals in **CC**, and then consider our solutions. In **CC** à la Curry, let $\mathsf{Nat} := \forall X.(X \to X) \to X \to X$. One can obtain a notion of *indexed iterator* by $\mathsf{It} := \lambda x.\lambda f.\lambda a.x\ f\ a$ and $\mathsf{It} : \forall X.\Pi x : \mathsf{Nat}.(X \to X) \to X \to X$. Thus we have $\mathsf{It}\ \bar{n} =_\beta \lambda f.\lambda a.\bar{n}\ f\ a =_\beta \lambda f.\lambda a.\underbrace{f(f(f...(f}_{n}\ a)...))$. One may want to know if we can obtain a finer version, namely, the induction principle-Ind such that:

$$\mathsf{Ind} : \forall P : \mathsf{Nat} \to *.\Pi x : \mathsf{Nat}.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to P\ x$$

Let us try to construct such Ind. First observe the following beta-equalities and typings:

$$\mathsf{Ind}\ \bar{0} =_\beta \lambda f.\lambda a.a$$

$$\mathsf{Ind}\ \bar{0} : (\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to P\ \bar{0}$$

$$\mathsf{Ind}\ \bar{n} =_\beta \lambda f.\lambda a.\underbrace{f\ \overline{n-1}(...f\ \bar{1}\ (f}_{n>0}\ \bar{0}\ a))$$

$$\mathsf{Ind}\ \bar{n} : (\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to P\ \bar{n}$$

$$\text{with } f : \Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y)), a : P\ \bar{0}$$

These equalities suggest that $\mathsf{Ind} := \lambda x.\lambda f.\lambda a.x\ f\ a$, using Parigot numerals [40]:

$$\bar{0} := \lambda s.\lambda z.z$$

$$\bar{n} := \lambda s.\lambda z.s\ \overline{n-1}\ (\overline{n-1}\ s\ z)$$

Each numeral corresponds to its terminating recursor.

Now, let us try to type these lambda numerals. It is reasonable to assign

$s : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y))$ and $z : P\ \bar{0}$. Thus we have the following typing relations:

$$\bar{0} : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{0}$$

$$\bar{1} : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{1}$$

$$\bar{n} : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{n}$$

So we want to define Nat to be something like:

$$\forall P : \mathsf{Nat} \to *.\Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{n} \text{ for any } \bar{n}.$$

Two problems arise with this scheme of encoding. The first problem involves recursiveness. The definiens of Nat contains Nat and $\mathsf{S}, \bar{0}$, while the type of S is $\mathsf{Nat} \to \mathsf{Nat}$ and the type of $\bar{0}$ is Nat. So the typing of Nat will be mutually recursive. Observe that the recursive occurrences of Nat are all at the type-annotated positions; i.e., the right side of the ":".

Note that the subdata of $\bar{n}$ is responsible for one recursive occurrence of Nat, namely, $\Pi y : \mathsf{Nat}$. If one never computes with the subdata, then these numerals will behave just like Church numerals. This inspires us to use Miquel's implicit product [36]. In this case, we want to redefine Nat to be something like:

$$\forall P : \mathsf{Nat} \to *.\forall y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{n} \text{ for any } \bar{n}.$$

Here $\forall y : \mathsf{Nat}$ is the implicit product. Now our notion of numerals are exactly Church numerals instead of Parigot numerals. Even better, this definition of Nat can be erased to $\mathbf{F}_\omega$. Since $\mathbf{F}_\omega$'s types do not have dependency on terms, $P : \mathsf{Nat} \to *$ will get erased to $P : *$. It is known that one can also erase the implicit product [3]. The erasure of Nat will be $\forall P : *.(P \to P) \to P \to P$, which is the definition of Nat in

$\mathbf{F}_\omega$.

The second problem is about quantification. We want to define a type Nat for any $\bar{n}$, but right now what we really have is one Nat for each numeral $\bar{n}$. We solve this problem by introducing a new type construct $\iota x.T$ called a *self type*. This allows us to make this definition (for Church-encoded naturals):

$$\mathsf{Nat} := \iota x.\forall P : \mathsf{Nat} \to *.\forall y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ x$$

We require that the self type can only be instantiated/generalized by its own subject, so we add the following two rules:

$$\frac{\Gamma \vdash t : [t/x]T}{\Gamma \vdash t : \iota x.T}\ \mathit{selfGen} \qquad \frac{\Gamma \vdash t : \iota x.T}{\Gamma \vdash t : [t/x]T}\ \mathit{selfInst}$$

We have the following inferences[1]:

$$\frac{\bar{n} : \forall P : \mathsf{Nat} \to *.\forall y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{n}}{\overline{\bar{n} : \iota x.\forall P : \mathsf{Nat} \to *.\forall y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ x}}$$

5.2.2 The Notion of Contradiction

In **CC** à la Curry, it is customary to use $\forall X : *.X$ as the notion of contradiction, since an inhabitant of the type $\forall X : *.X$ will inhabit any type, so the law of explosion is subsumed by the type $\forall X : *.X$. However, this notion of contradiction is too strong to be useful. Let $t =_A t'$ denote $\forall C : A \to *.C\ t \to C\ t'$ with $t, t' : A$. Then $0 =_{\mathsf{Nat}} 1$ can be expanded to $\forall C : \mathsf{Nat} \to *.C\ 0 \to C\ 1$ (0 is Leibniz equals to 1). One can not derive a proof for $(\forall C : \mathsf{Nat} \to *.C\ 0 \to C\ 1) \to \forall X : *.X$, because the erasure of $(\forall C : \mathsf{Nat} \to *.C\ 0 \to C\ 1) \to \forall X : *.X$ in System **F** would be $(\forall C : *.C \to C) \to \forall X : *.X$, and we know that $\forall C : *.C \to C$ is inhabited. So the

[1]The double bar means that the converse of the inference also holds.

inhabitation of $(\forall C : \mathsf{Nat} \to *.C\ 0 \to C\ 1) \to \forall X : *.X$ will imply the inhabitation of $\forall X : *.X$ in System **F**, which does not hold. If we take Leibniz equality and use $\forall X : *.X$ as contradiction, then we can not prove any negative results about equality.

On the other hand, an equational theory is considered inconsistent if $a = b$ for all term a and b. So we propose to use $\forall A : *.\Pi x : A.\Pi y : A.x =_A y$ as the notion of contradiction in **CC**. We first want to make sure it is uninhabited. The way to argue that is first assume it is inhabited by t. Since **CC** is strongly normalizing, the normal form of t must be of the form[2] $[\lambda A : *.]\lambda x[: A].\lambda y[: A].[\lambda C : A \to *].\lambda z[: C\ x].n$ for some normal term n with type $C\ y$, but we know that there is no combination of x, y, z to make a term of type $C\ y$. So the type $\forall A : *.\Pi x : A.\Pi y : A.\forall C : A \to *.Cx \to Cy$ is uninhabited. We can then prove the following theorem.

Theorem 5. *$0 = 1 \to \bot$ is inhabited in* **CC**, *where $\bot := \forall A : *.\Pi x : A.\Pi y : A.\forall C : A \to *.C\ x \to C\ y$, $0 := \lambda s.\lambda z.z$, $1 := \lambda s.\lambda z.s\ z$.*

Proof. Assume $\mathsf{Nat} := \forall B : *.(B \to B) \to B \to B$. Let $\Gamma = a : (\forall D : \mathsf{Nat} \to *.D0 \to D1), A : *, x : A, y : A, C : A \to *, c : C\ x$. We want to construct a term of type $C\ y$. Let $F := \lambda n[: \mathsf{Nat}].n\ [A]\ (\lambda q[: A].y)x$. Note that $F : \mathsf{Nat} \to A$. We know that $F0 =_\beta x$ and $F1 =_\beta y$. So we can indeed convert the type of c from Cx to $C\ (F0)$. And then we instantiate the D in $\forall D : \mathsf{Nat} \to *.D0 \to D1$ with $\lambda x[: \mathsf{Nat}].C\ (Fx)$. So we have $C\ (F0) \to C\ (F1)$ as the type of a. So $a\ c : C(F1)$, which means $a\ c : Cy$. So we just show how to inhabit $0 = 1 \to \bot$ in **CC**.

[2]We use square brackets [ ] to show annotations that are not present in the inhabiting lambda term in Curry-style System **F**.

□

Once $\bot$ is derived, one can not distinguish the domain of individuals. Note that this notion of contradiction does not subsume law of explosion.

5.3 System S

Definition 45 (Syntax)**.**

Terms $t \ ::= \ x \mid \lambda x.t \mid tt'$

Types $T \ ::= \ X \mid \forall X : \kappa.T \mid \Pi x : T_1.T_2 \mid \forall x : T_1.T_2 \mid \iota x.T \mid T\ t \mid \lambda X.T \mid \lambda x.T \mid T_1 T_2$

Kinds $\kappa \ ::= \ * \mid \Pi x : T.\kappa \mid \Pi X : \kappa'.\kappa$

Context $\Gamma \ ::= \ \cdot \mid \Gamma, x : T \mid \Gamma, X : \kappa \mid \Gamma, \mu$

Closure $\mu \ ::= \ \{(x_i : S_i) \mapsto t_i\}_{i \in N} \cup \{(X_i : \kappa_i) \mapsto T_i\}_{i \in M}$

Closures. For $\{(x_i : S_i) \mapsto t_i\}_{i \in N}$, we mean the term variable x_i of type S_i is defined to be t_i for some $i \in N$; similarly for $\{(X_i : \kappa_i) \mapsto T_i\}_{i \in M}$.

Legal positions for recursion in closures. For $\{(x_i : S_i) \mapsto t_i\}_{i \in N}$, we do not allow any recursive (or mutually recursive) definitions. For $\{(X_i : \kappa_i) \mapsto T_i\}_{i \in M}$, we only allow singly recursive type definitions, but not mutually recursive ones. This is not a fundamental limitation of the approach; it is just for simplicity of the normalization argument. The recursive occurrences of type variables can only be at positive or erased positions. Erased positions, following the erasure function we will see in Section 5.5.1, are those in kinds or in the types for $\forall$-bound variables.

Variable restrictions for closures. Let $\mathrm{FV}(e)$ denote the set of free term variables in expression e (either term, type, or kind), and let $\mathrm{FVar}(T)$ denote the set of free type variables in type T. Then for $\{(x_i : S_i) \mapsto t_i\}_{i \in N} \cup \{(X_i : \kappa_i) \mapsto T_i\}_{i \in M}$, we make the simplifying assumption that for any $1 \leq i \leq n$, $\mathrm{FV}(t_i) = \emptyset$. Also, for any $1 \leq i \leq m$, we require $\mathrm{FV}(T_i) \subseteq \mathrm{dom}(\mu)$, and $\mathrm{FVar}(\mathrm{T_i}) \subseteq \{X_i\}$. All our examples below satisfy these conditions.

Notation for accessing closures. $(t_i : S_i) \in \mu$ means $(x_i : S_i) \mapsto t_i \in \mu$ and $(T_i : \kappa_i) \in \mu$ means $(X_i : \kappa_i) \mapsto T_i \in \mu$. Also, $x_i \mapsto t_i \in \mu$ means $(x_i : S_i) \mapsto t_i \in \mu$ for some S_i and $X_i \mapsto T_i \in \mu$ means $(X_i : \kappa_i) \mapsto T_i \in \mu$ for some κ_i.

Well-formed annotated closures. $\Gamma \vdash \mu\ \mathsf{ok}$ stands for $\{\Gamma, \mu \vdash t_j : T_j\}_{(t_j:T_j)\in\mu}$ and $\{\Gamma, \mu \vdash T_j : \kappa_j\}_{(T_j:\kappa_j)\in\mu}$. In other words, the defining expressions in closures must be typable with respect to the context and the entire closure.

Notation for equivalence. $\cong$ is the congruence closure of $\to_\beta$.

Self type formation. Typing and kinding do not depend on well-formedness of the context, so the self type formation rule *self* is not circular.

Well-formed Contexts $\boxed{\Gamma \vdash \mathsf{wf}}$

$$\frac{}{\cdot \vdash \mathsf{wf}} \qquad \frac{\Gamma \vdash \mathsf{wf} \quad \Gamma \vdash T : *}{\Gamma, x : T \vdash \mathsf{wf}} \qquad \frac{\Gamma \vdash \mathsf{wf} \quad \Gamma \vdash \kappa : \Box}{\Gamma, X : \kappa \vdash \mathsf{wf}} \qquad \frac{\Gamma \vdash \mathsf{wf} \quad \Gamma \vdash \mu\ \mathsf{ok}}{\Gamma, \mu \vdash \mathsf{wf}}$$

Well-formed Kinds $\boxed{\Gamma \vdash \kappa : \Box}$

$$\frac{}{\Gamma \vdash * : \Box} \qquad \frac{\Gamma, X : \kappa' \vdash \kappa : \Box \quad \Gamma \vdash \kappa' : \Box}{\Gamma \vdash \Pi X : \kappa'.\kappa : \Box} \qquad \frac{\Gamma, x : T \vdash \kappa : \Box \quad \Gamma \vdash T : *}{\Gamma \vdash \Pi x : T.\kappa : \Box}$$

Kinding $\boxed{\Gamma \vdash T : \kappa}$

$$\frac{(X:\kappa)\in\Gamma}{\Gamma\vdash X:\kappa} \qquad \frac{\Gamma\vdash T:\kappa \quad \Gamma\vdash\kappa\cong\kappa' \quad \Gamma\vdash\kappa':\square}{\Gamma\vdash T:\kappa'}$$

$$\frac{\Gamma\vdash T_1:* \quad \Gamma,x:T_1\vdash T_2:*}{\Gamma\vdash \Pi x:T_1.T_2:*} \qquad \frac{\Gamma,X:\kappa\vdash T:* \quad \Gamma\vdash\kappa:\square}{\Gamma\vdash \forall X:\kappa.T:*}$$

$$\frac{\Gamma,x:T_1\vdash T_2:* \quad \Gamma\vdash T_1:*}{\Gamma\vdash \forall x:T_1.T_2:*} \qquad \frac{\Gamma,x:\iota x.T\vdash T:*}{\Gamma\vdash \iota x.T:*}\ \textit{Self}$$

$$\frac{\Gamma,X:\kappa\vdash T:\kappa' \quad \Gamma\vdash\kappa:\square}{\Gamma\vdash \lambda X.T:\Pi X:\kappa.\kappa'} \qquad \frac{\Gamma,x:T'\vdash T:\kappa \quad \Gamma\vdash T':*}{\Gamma\vdash \lambda x.T:\Pi x:T'.\kappa}$$

$$\frac{\Gamma\vdash S:\Pi x:T.\kappa \quad \Gamma\vdash t:T}{\Gamma\vdash S\ t:[t/x]\kappa} \qquad \frac{\Gamma\vdash S:\Pi X:\kappa'.\kappa \quad \Gamma\vdash T:\kappa'}{\Gamma\vdash S\ T:[T/X]\kappa}$$

Typing $\boxed{\Gamma\vdash t:T}$

$$\frac{\Gamma\vdash t:T_1 \quad \Gamma\vdash T_1\cong T_2 \quad \Gamma\vdash T_2:*}{\Gamma\vdash t:T_2}\ \textit{Conv} \qquad \frac{(x:T)\in\Gamma}{\Gamma\vdash x:T}\ \textit{Var}$$

$$\frac{\Gamma\vdash t:[t/x]T \quad \Gamma\vdash \iota x.T:*}{\Gamma\vdash t:\iota x.T}\ \textit{SelfGen} \qquad \frac{\Gamma\vdash t:\iota x.T}{\Gamma\vdash t:[t/x]T}\ \textit{SelfInst}$$

$$\frac{\Gamma,x:T_1\vdash t:T_2 \quad \Gamma\vdash T_1:* \quad x\notin \mathrm{FV}(t)}{\Gamma\vdash t:\forall x:T_1.T_2}\ \textit{Indx} \qquad \frac{\Gamma\vdash t:\forall x:T_1.T_2 \quad \Gamma\vdash t':T_1}{\Gamma\vdash t:[t'/x]T_2}\ \textit{Dex}$$

$$\frac{\Gamma\vdash t:\Pi x:T_1.T_2 \quad \Gamma\vdash t':T_1}{\Gamma\vdash tt':[t'/x]T_2}\ \textit{App} \qquad \frac{\Gamma,X:\kappa\vdash t:T \quad \Gamma\vdash\kappa:\square}{\Gamma\vdash t:\forall X:\kappa.T}\ \textit{Poly}$$

$$\frac{\Gamma\vdash t:\forall X:\kappa.T \quad \Gamma\vdash T':\kappa}{\Gamma\vdash t:[T'/X]T}\ \textit{Inst} \qquad \frac{\Gamma,x:T_1\vdash t:T_2 \quad \Gamma\vdash T_1:*}{\Gamma\vdash \lambda x.t:\Pi x:T_1.T_2}\ \textit{Func}$$

Reductions $\boxed{\Gamma\vdash t\to_\beta t'}$, $\boxed{\Gamma\vdash T\to_\beta T'}$

$$\frac{(x\mapsto t)\in\Gamma}{\Gamma\vdash x\to_\beta t} \qquad \frac{}{\Gamma\vdash(\lambda x.t)t'\to_\beta [t'/x]t} \qquad \frac{(X\mapsto T)\in\Gamma}{\Gamma\vdash X\to_\beta T}$$

$$\frac{}{\Gamma\vdash(\lambda x.T)t\to_\beta [t/x]T} \qquad \frac{}{\Gamma\vdash(\lambda X.T)T'\to_\beta [T'/X]T}$$

5.4 Lambda Encodings in S

Now let us see some concrete examples of lambda encoding in **S**. For convenience, we write $T \to T'$ for $\Pi x : T.T'$ with $x \notin \mathrm{FV}(T')$, and similarly for kinds.

5.4.1 Natural Numbers

Definition 46 (Church Numerals)**.** *Let μ_c be the following closure:*

$(\mathsf{Nat} : *) \mapsto \iota x.\forall C : \mathsf{Nat} \to *.(\forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n)) \to C\ 0 \to C\ x$

$(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$

$(0 : \mathsf{Nat}) \mapsto \lambda s.\lambda z.z$

With $s : \forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n), z : C\ 0, n : \mathsf{Nat}$, we have $\mu_c \vdash \mathsf{wf}$ (using *selfGen* and *selfInst* rules). Also note that the μ_c satisfies the constraints on recursive definitions. Similarly, if we choose to use explicit product, then we can define Parigot numerals.

Definition 47 (Parigot Numerals)**.** *Let μ_p be the following closure:*

$(\mathsf{Nat} : *) \mapsto \iota x.\forall C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n)) \to C\ 0 \to C\ x$

$(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda s.\lambda z.s\ n\ (n\ s\ z)$

$(0 : \mathsf{Nat}) \mapsto \lambda s.\lambda z.z$

Note that the recursive occurences of Nat in Parigot numerals are at positive positions. The rest of the examples are about Church numerals, but a similar development can be carried out with Parigot numerals.

Theorem 6 (Induction Principle)**.**

$\mu_c \vdash \mathsf{Ind} : \forall C : \mathsf{Nat} \to *.(\forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n)) \to C\ 0 \to \Pi n : \mathsf{Nat}.C\ n$

where $\mathsf{Ind} := \lambda s.\lambda z.\lambda n.n\ s\ z$

with $s : \forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n), z : C\ 0, n : \mathsf{Nat}.$

Proof. Let $\Gamma = \mu_c, C : \mathsf{Nat} \to *, s : \forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n), z : C\ 0, n : \mathsf{Nat}$. Since $n : \mathsf{Nat}$, by *selfInst*, $n : \forall C : \mathsf{Nat} \to *.(\forall y : \mathsf{Nat}.C\ y \to C\ (\mathsf{S}\ y)) \to C\ 0 \to C\ n$. Thus $n\ s\ z : C\ n$. □

It is worth noting that it is really the definition of Nat and the *selfInst* rule that give us the induction principle, which is not derivable in **CC** [14].

Definition 48 (Addition)**.** $m + n := \mathsf{Ind}\ \mathsf{S}\ n\ m$

One can check that $\mu_c \vdash + : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ by instantiating the C in the type of Ind by $\lambda y.\mathsf{Nat}$, then the type of Ind is $(\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})$.

Definition 49 (Leibniz's Equality)**.**

$\mathsf{Eq} := \lambda A[: *].\lambda x[: A].\lambda y[: A].\forall C : A \to *.C\ x \to C\ y.$

Note that we use $x =_A y$ to denote $\mathsf{Eq}\ A\ x\ y$. We often write $t = t'$ when the type is clear. One can check that if $\vdash A : *$ and $\vdash x, y : A$, then $\vdash x =_A y : *$.

Theorem 7. $\mu_c \vdash \Pi x : \mathsf{Nat}.x + 0 =_{\mathsf{Nat}} x$

Proof. We prove this by induction. We instantiate C in the type of Ind with $\lambda n.(n + 0) =_{\mathsf{Nat}} n$. So by beta reduction at type level, we have $(\forall n : \mathsf{Nat}.(n + 0 =_{\mathsf{Nat}} n) \to ((\mathsf{S}\ n) + 0 =_{\mathsf{Nat}} \mathsf{S}\ n)) \to 0 + 0 =_{\mathsf{Nat}} 0 \to \Pi n : \mathsf{Nat}.n + 0 =_{\mathsf{Nat}} n$. So for the base case, we need to show $0 + 0 =_{\mathsf{Nat}} 0$, which is easy. For the step case, we assume $n + 0 =_{\mathsf{Nat}} n$ (Induction Hypothesis), and want to show $(\mathsf{S}\ n) + 0 =_{\mathsf{Nat}} \mathsf{S}\ n$. Since

$(\mathsf{S}\ n) + 0 \to_\beta \mathsf{S}\ (n\ \mathsf{S}\ 0) =_\beta \mathsf{S}(n+0)$, by congruence on the induction hypothesis, we have $(\mathsf{S}\ n) + 0 =_{\mathsf{Nat}} \mathsf{S}\ n$. Thus $\Pi x : \mathsf{Nat}. x + 0 =_{\mathsf{Nat}} x$. □

5.4.2 Vector Encoding

Definition 50 (Vector)**.** *Let μ_v be the following definitions:*

$(\mathsf{vec} : * \to \mathsf{Nat} \to *) \mapsto$

$$\lambda U : *.\lambda n : \mathsf{Nat}.\iota x.\forall C : \Pi p : \mathsf{Nat}.\mathsf{vec}\ U\ p \to *.$$

$$(\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(C\ m\ y\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ y)))$$

$$\to C\ 0\ \mathsf{nil} \to C\ n\ x$$

$(\mathsf{nil} : \forall U : *.\mathsf{vec}\ U\ 0) \mapsto \lambda y.\lambda x.x$

$(\mathsf{cons} : \Pi n : \mathsf{Nat}.\forall U : *.U \to \mathsf{vec}\ U\ n \to \mathsf{vec}\ U\ (\mathsf{S}\ n)) \mapsto \lambda n.\lambda v.\lambda l.\lambda y.\lambda x.y\ n\ v\ (l\ y\ x)$

where $n : \mathsf{Nat}, v : U, l : \mathsf{vec}\ U\ n, y : \Pi m : \mathsf{Nat}.\Pi u : U.\forall z : \mathsf{vec}\ U\ m.(C\ m\ z\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ z)), x : C\ 0\ \mathsf{nil}$.

Typing: It is easy to see that nil is typable to $\forall U : *.\mathsf{vec}\ U\ 0$. Now we show how cons is typable to $\Pi n : \mathsf{Nat}.\forall U : *.U \to \mathsf{vec}\ U\ n \to \mathsf{vec}\ U\ (\mathsf{S}\ n)$. We can see that $l\ y\ x : C\ n\ l$ (using *selfinst* on l). After the instantiation with l, the type of $y\ n\ v$ is $C\ n\ l \to C\ (\mathsf{S}\ n)\ (\mathsf{cons}\ n\ v\ l)$. So $y\ n\ v\ (l\ y\ x) : C\ (\mathsf{S}\ n)\ (\mathsf{cons}\ n\ v\ l)$. So $\lambda y.\lambda x.y\ n\ v\ (l\ y\ x) : \Pi C : (\mathsf{Nat} \to \mathsf{vec}\ U\ p \to *).(\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(C\ m\ y\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ y))) \to C\ 0\ \mathsf{nil} \to C\ (\mathsf{S}\ n)\ (\lambda y.\lambda x.y\ n\ v\ (l\ y\ x))$. So by *selfGen*, we have $\lambda y.\lambda x.y\ n\ v\ (l\ y\ x) : \mathsf{vec}\ U (\mathsf{S}\ n)$. Thus $\mathsf{cons} : \Pi n : \mathsf{Nat}.\forall U : *.U \to \mathsf{vec}\ U\ n \to \mathsf{vec}\ U\ (\mathsf{S}\ n)$.

Definition 51 (Induction Principle for Vector)**.**

$\mu_v \vdash \mathsf{Ind} : \forall U : *.\Pi n : \mathsf{Nat}.\forall C : \mathsf{Nat} \to \mathsf{vec}\ U\ p \to *.$

$(\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(C\ m\ y\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ y)))$

$\to C\ 0\ \mathsf{nil} \to \Pi x : \mathsf{vec}\ U\ n.(C\ n\ x)$

where $\mathsf{Ind} := \lambda n.\lambda s.\lambda z.\lambda x.x\ s\ z$

$n : \mathsf{Nat}, s : \forall C : (\mathsf{Nat} \to \mathsf{vec}\ U\ p \to *).(\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(C\ m\ y\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ y))), z : C\ 0\ \mathsf{nil}, x : \mathsf{vec}\ U\ n.$

Definition 52 (Append). $\mu_v \vdash \mathsf{app} : \forall U : *.\Pi n_1 : \mathsf{Nat}.\Pi n_2 : \mathsf{Nat}.\mathsf{vec}\ U\ n_1 \to \mathsf{vec}\ U\ n_2 \to \mathsf{vec}\ U\ (n_1 + n_2)$

where $\mathsf{app} := \lambda n_1.\lambda n_2.\lambda l_1.\lambda l_2.(\mathsf{Ind}\ n_1)\ (\lambda n.\lambda x.\lambda v.\mathsf{cons}\ (n + n_2)\ x\ v)\ l_2\ l_1.$

Typing: We want to show $\mathsf{app} : \forall U : *.\Pi n_1 : \mathsf{Nat}.\Pi n_2 : \mathsf{Nat}.\mathsf{vec}\ U\ n_1 \to \mathsf{vec}\ U\ n_2 \to \mathsf{vec}\ U\ (n_1 + n_2)$. Observe that $\lambda n.\lambda x.\lambda v.\mathsf{cons}(n + n_2)\ x\ v : \Pi n : \mathsf{Nat}.\Pi x : U.\mathsf{vec}\ U\ (n + n_2) \to \mathsf{vec}\ U\ (n + n_2 + 1)$. We instantiate $C := \lambda y.(\lambda x.\mathsf{vec}\ U\ (y + n_2))$, where x free over $\mathsf{vec}\ U\ (y + n_2)$, in $\mathsf{Ind}\ n_1$. By beta reductions, we get $\mathsf{Ind}\ n_1 : (\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(\mathsf{vec}\ U\ (m + n_2) \to \mathsf{vec}\ U\ ((\mathsf{S}\ m) + n_2)) \to \mathsf{vec}\ U\ (0 + n_2) \to \Pi x : \mathsf{vec}\ U\ n_1.\mathsf{vec}\ U\ (n_1 + n_2)$.

So $(\mathsf{Ind}\ n_1)\ (\lambda n.\lambda x.\lambda v.\mathsf{cons}(n+n_2)\ x\ v) : \mathsf{vec}\ U\ (0+n_2) \to \Pi x : \mathsf{vec}\ U\ n_1.\mathsf{vec}\ U\ (n_1+n_2)$. We assume $l_1 : \mathsf{vec}\ U\ n_1, l_2 : \mathsf{vec}\ U\ n_2$. Thus $(\mathsf{Ind}\ n_1)\ (\lambda n.\lambda x.\lambda v.\mathsf{cons}(n+n_2)\ x\ v)\ l_2\ l_1 : \mathsf{vec}\ U\ (n_1 + n_2)$.

5.5 Metatheory

We first outline the erasure from $\mathbf{S}$ to $\mathbf{F}_\omega$ with positive recursive definitions, which shows the strong normalization of $\mathbf{S}$. We also prove type preservation for $\mathbf{S}$, which involves *confluence analysis* (Section 5.5.2) and *morph analysis* (Section 5.5.3).

5.5.1 Strong Normalization

We prove strong normalization of $\mathbf{S}$ through the strong normalization of $\mathbf{F}_\omega$ with positive recursive definitions. We first define the syntax for $\mathbf{F}_\omega$ with positive recursive definitions.

Definition 53 (Syntax for $\mathbf{F}_\omega$ with positive definitions)**.**

Terms $t\ ::=\ x \mid \lambda x.t \mid tt'$

Kinds $\kappa\ ::=\ * \mid \kappa' \to \kappa$

Types $T^\kappa\ ::=\ X^\kappa \mid (\forall X^\kappa.T^*)^* \mid (T_1^* \to T_2^*)^* \mid (\lambda X^{\kappa_1}.T^{\kappa_2})^{\kappa_1 \to \kappa_2} \mid (T_1^{\kappa_1\to\kappa_2} T_2^{\kappa_1})^{\kappa_2}$

Context $\Gamma\ ::=\ \cdot \mid \Gamma, x : T^\kappa \mid \Gamma, \mu$

Definitions $\mu\ ::=\ \{(x_i : S_i^\kappa) \mapsto t_i\}_{i\in N} \cup \{X_i^\kappa \mapsto T_i^\kappa\}_{i\in M}$

Term definitions $\rho\ ::=\ \{x_i \mapsto t_i\}_{i\in N}$

Note that for every $x \mapsto t, X^\kappa \mapsto T^\kappa \in \mu$, we require $\mathrm{FV}(t) = \emptyset$ and $\mathrm{FVar}(T^\kappa) \subseteq \{X^\kappa\}$; and the X^κ can only occur at the positive position in T^κ, no mutually recusive definitions are allowed. We elide the typing rules for space reason. We adopt kind-annotated types to obtain a clearer interpretation of types. e.g. with kind annotation, we do not need to worry about interpretation for ill-formed types like $(\lambda X.X) \to (\lambda X.X)$.

Definition 54 (Erasure for kinds)**.** *We define a function F maps kinds in* $\mathbf{S}$ *to kinds in* $\mathbf{F}_\omega$ *with positive definitions.*

$$F(*)\ := *$$

$$F(\Pi x : T.\kappa)\ := F(\kappa)$$

$$F(\Pi X : \kappa'.\kappa)\ := F(\kappa') \to F(\kappa)$$

Definition 55 (Erasure relation)**.** *We define relation* $\Gamma \vdash T \triangleright T'^{\kappa}$ *(intuitively, it means that type* T *can be erased to* T'^{κ} *under the context* Γ*), where* T, Γ *are types and context in* $\mathbf{S}$*,* T'^{κ} *is a type in* $\mathbf{F}_\omega$ *with positive definitions.*

$$\frac{F(\kappa') = \kappa \quad (X : \kappa') \in \Gamma}{\Gamma \vdash X \triangleright X^{\kappa}} \qquad \frac{\Gamma \vdash T \triangleright T_1^{\kappa}}{\Gamma \vdash \iota x.T \triangleright T_1^{\kappa}}$$

$$\frac{\Gamma, X : \kappa \vdash T \triangleright T_1^{*}}{\Gamma \vdash \forall X : \kappa.T \triangleright (\forall X^{F(\kappa)}.T_1^{*})^{*}} \qquad \frac{\Gamma \vdash T_1 \triangleright T_a^{*} \quad \Gamma \vdash T_2 \triangleright T_b^{*}}{\Gamma \vdash \Pi x : T_1.T_2 \triangleright (T_a^{*} \to T_b^{*})^{*}}$$

$$\frac{\Gamma \vdash T_2 \triangleright T^{\kappa}}{\Gamma \vdash \forall x : T_1.T_2 \triangleright T^{\kappa}} \qquad \frac{\Gamma \vdash T_1 \triangleright T_a^{\kappa_1 \to \kappa_2} \quad \Gamma \vdash T_b^{\kappa_1}}{\Gamma \vdash T_1 T_2 \triangleright (T_a^{\kappa_1 \to \kappa_2} T_b^{\kappa_1})^{\kappa_2}}$$

$$\frac{\Gamma, X : \kappa \vdash T \triangleright T_a^{\kappa'}}{\Gamma \vdash \lambda X.T \triangleright (\lambda X^{F(\kappa)}.T_a^{\kappa'})^{\kappa \to \kappa'}} \qquad \frac{\Gamma \vdash T \triangleright T_1^{\kappa}}{\Gamma \vdash T\ t \triangleright T_1^{\kappa}}$$

$$\frac{\Gamma \vdash T \triangleright T_1^{\kappa}}{\Gamma \vdash \lambda x.T \triangleright T_1^{\kappa}}$$

Definition 56 (Erasure for Context)**.** *We define relation* $\Gamma \triangleright \Gamma'$ *inductively.*

$$\frac{\Gamma \vdash T \triangleright T_a^{F(\kappa)} \quad \Gamma \triangleright \Gamma'}{\Gamma, (X : \kappa) \mapsto T \triangleright \Gamma', X^{F(\kappa)} \mapsto T_a^{F(\kappa)}} \qquad \frac{\Gamma \vdash \Gamma'}{\Gamma, X : \kappa \triangleright \Gamma'} \qquad \frac{}{\cdot \triangleright \cdot}$$

$$\frac{\Gamma \vdash T \triangleright T_a^{\kappa} \quad \Gamma \triangleright \Gamma'}{\Gamma, (x : T) \mapsto t \triangleright \Gamma', x : T_a^{\kappa} \mapsto t} \qquad \frac{\Gamma \vdash T \triangleright T_a^{\kappa} \quad \Gamma \triangleright \Gamma'}{\Gamma, x : T \triangleright \Gamma', x : T_a^{\kappa}}$$

Theorem 8 (Erasure Theorem)**.**

1. *If* $\Gamma \vdash T : \kappa$*, then there exists a* $T_a^{F(\kappa)}$ *such that* $\Gamma \vdash T \triangleright T_a^{F(\kappa)}$*.*

2. *If* $\Gamma \vdash t : T$ *and* $\Gamma \vdash \mathsf{wf}$*, then there exist* T_a^{*} *and* Γ' *such that* $\Gamma \vdash T \triangleright T_a^{*}$*,* $\Gamma \triangleright \Gamma'$ *and* $\Gamma' \vdash t : T_a^{*}$*.*

Now that we obtained an erasure from $\mathbf{S}$ to $\mathbf{F}_\omega$ with positive definitions. We continue to show latter is strongly normalizing. The development below is in $\mathbf{F}_\omega$ with positive definitions. Let $\mathfrak{R}_\rho$ be the set of all reducibility candidates[3]. Let σ be a mapping between type variable of kind κ to element of $\rho[\![\kappa]\!]$.

Definition 57.

- $\rho[\![*]\!] := \mathfrak{R}_\rho$.
- $\rho[\![\kappa \to \kappa']\!] := \{f \mid \forall a \in \rho[\![\kappa]\!], f(a) \in \rho[\![\kappa']\!]\}$.
- $\rho[\![X^\kappa]\!]_\sigma := \sigma(X^\kappa)$.
- $\rho[\![(T_1^* \to T_2^*)^*]\!]_\sigma := \{t \mid \forall u. \in \rho[\![T_1^*]\!]_\sigma, tu \in \rho[\![T_2^*]\!]_\sigma\}$.
- $\rho[\![(\forall X^\kappa.T^*)^*]\!]_\sigma := \bigcap_{f\in\rho[\![\kappa]\!]} \rho[\![T^*]\!]_{\sigma[f/X]}$.
- $\rho[\![(\lambda X^{\kappa'}.T^\kappa)^{\kappa'\to\kappa}]\!]_\sigma := f$ *where* f *is the map* $a \mapsto \rho[\![T^\kappa]\!]_{\sigma[a/X]}$ *for any* $a \in \rho[\![\kappa']\!]$.
- $\rho[\![(T_1^{\kappa'\to\kappa} T_2^{\kappa'})^\kappa]\!]_\sigma := \rho[\![T_1^{\kappa'\to\kappa}]\!]_\sigma(\rho[\![T_2^{\kappa'}]\!]_\sigma)$.

Let $|\cdot|$ be a function that retrieves all the term definitions from the context Γ.

Definition 58. *Let* $\rho = |\Gamma|$, *and* $\mathrm{FVar}(\Gamma)$ *be the set of free type variables in* Γ. *We define* $\sigma \in \rho[\![\Gamma]\!]$ *if* $\sigma(X^\kappa) \in \rho[\![\kappa]\!]$ *for undefined variable* X^κ*; and* $\sigma(X^\kappa) = \mathrm{lfp}(b \mapsto \rho[\![T^\kappa]\!]_{\sigma[b/X^\kappa]})$ *for* $b \in \rho[\![\kappa]\!]$ *if* $X^\kappa \mapsto T^\kappa \in \Gamma$.

Note that the least fix point operation in $\mathrm{lfp}(b \mapsto \rho[\![T^\kappa]\!]_{\sigma[b/X^\kappa]})$ is defined since we can extend the complete lattice of reducibility candidate to complete lattice $(\rho[\![\kappa]\!], \subseteq_\kappa, \cap_\kappa)$.

[3]The notion of reducibility candidate here slightly extends the standard one to handle definitional reduction: $\rho \vdash x \to_\beta t$, where $x \mapsto t \in \rho$. So it is parametrized by ρ.

Definition 59. *Let $\rho = |\Gamma|$ and $\sigma \in \rho[\![\Gamma]\!]$. We define the relation $\delta \in \rho[\![\Gamma]\!]$ inductively:*

$$\frac{}{\cdot \in \rho[\![\cdot]\!]} \qquad \frac{\delta \in \rho[\![\Gamma]\!] \quad t \in \rho[\![T^\kappa]\!]_\sigma}{\delta[t/x] \in \rho[\![\Gamma, x : T^\kappa]\!]} \qquad \frac{\delta \in \Gamma}{\delta \in \rho[\![\Gamma, (x : T^\kappa) \mapsto t]\!]}$$

Theorem 9 (Soundness theorem)**.** *Let $\rho = |\Gamma|$. If $\Gamma \vdash t : T^*$ and $\Gamma \vdash \mathsf{wf}$, then for any $\sigma, \delta \in \rho[\![\Gamma]\!]$, we have $\delta t \in \rho[\![T^*]\!]_\sigma$, with $\rho[\![T^*]\!]_\sigma \in \mathfrak{R}_\rho$.*

Theorem 8 and 9 imply all the typable term in **S** is strongly normalizing.

5.5.2 Confluence Analysis

The complications of proving type preservation are due to several rules which are not syntax-directed. To prove type preservation, one needs to ensure that if $\Pi x : T.T'$ can be transformed to $\Pi x : T_1.T_2$, then it must be the case that T can be transformed to T_1 and T' can be transformed to T_2. This is why we need to show confluence for type-level reduction. We first observe that the *selfGen* rule and *selfInst* rule are mutually inverse, and model the change of self type by the following reduction relation.

Definition 60.

$\Gamma \vdash T_1 \to_\iota T_2$ if $T_1 \equiv \iota x.T'$ and $T_2 \equiv [t/x]T'$ for some fix term t.

Note that $\to_\iota$ models the *selfInst* rule, $\to_\iota^{-1}$ models the *selfGen* rule. Importantly, the notion of ι-reduction does not include congruence; that is, we do not allow reduction rules like if $T \to_\iota T'$, then $\lambda x.T \to_\iota \lambda x.T'$. The purpose of ι-reduction is to emulate the typing rule *selfInst* and *selfGen*.

We first show confluence of $\to_\beta$ by applying the standard Tait-Martin Löf method, and then apply Hindley-Rossen's commutativity theorem to show $\to_\iota$ commutes with $\to_\beta$. We use $\to^*$ to denote the reflexive symmetric transitive closure of $\to$.

Lemma 22. *$\to_\beta$ is confluent.*

Definition 61 (Commutativity)**.** *Let $\to_1, \to_2$ be two notions of reduction. Then $\to_1$ commutes with $\to_2$ iff $\leftarrow_1 \cdot \to_2 \subseteq \to_1 \cdot \leftarrow_2$.*

Proposition 1. *Let $\to_1, \to_2$ be two notions of reduction. Suppose both $\to_1$ and $\to_2$ are confluent, and $\to_1^*$ commutes with $\to_2^*$. Then $\to_1 \cup \to_2$ is confluent.*

Lemma 23. *$\to_\beta$ commutes with $\to_\iota$. Thus $\to_{\beta,\iota}$ is confluent, where $\to_{\beta,\iota} = \to_\beta \cup \to_\iota$.*

Theorem 10 (ι-elimination)**.** *If $\Gamma \vdash \Pi x : T_1.T_2 =_{\beta,\iota} \Pi x : T_1'.T_2'$, then $\Gamma \vdash T_1 =_\beta T_1'$ and $\Gamma \vdash T_2 =_\beta T_2'$.*

Proof. If $\Gamma \vdash \Pi x : T_1.T_2 =_{\beta,\iota} \Pi x : T_1'.T_2'$, then by the confluence of $\to_{\beta,\iota}$, there exists a T such that $\Gamma \vdash \Pi x : T_1.T_2 \to_{\iota,\beta}^* T$ and $\Gamma \vdash \Pi x : T_1'.T_2' \to_{\iota,\beta}^* T$. Since all the reductions on $\Pi x : T_1.T_2$ preserve the structure of the dependent type, one will never have a chance to use $\to_\iota$-reduction, thus $\Gamma \vdash \Pi x : T_1.T_2 \to_\beta^* T$ and $\Gamma \vdash \Pi x : T_1'.T_2' \to_\beta^* T$. So T must be of the form $\Pi x : T_3.T_4$. And $\Gamma \vdash T_1 \to_\beta^* T_3$, $\Gamma \vdash T_1' \to_\beta^* T_3$, $\Gamma \vdash T_2 \to_\beta^* T_4$ and $\Gamma \vdash T_2' \to_\beta^* T_4$. Finally, we have $\Gamma \vdash T_1 =_\beta T_1'$ and $\Gamma \vdash T_2 =_\beta T_2'$.

□

5.5.3 Morph Analysis

The methods of the previous section are not suitable for dealing with implicit polymorphism, since as a reduction relation, polymorphic instantiation is not confluent. For example, $\forall X : \kappa.X$ can be instantiated either to T or to $T \to T$. The only known syntactic method (to our knowledge) to deal with preservation proof for Curry-style System **F** is Barendregt's method [6]. We will extend his method to handle the instantiation of $\forall x : T.T'$.

Definition 62 (Morphing Relations)**.**

- $([\Gamma], T_1) \to_i ([\Gamma], T_2)$ *if* $T_1 \equiv \forall X : \kappa.T'$ *and* $T_2 \equiv [T/X]T'$ *for some* T *such that* $\Gamma \vdash T : \kappa$.

- $([\Gamma, X : \kappa], T_1) \to_g ([\Gamma], T_2)$ *if* $T_2 \equiv \forall X : \kappa.T_1$ *and* $\Gamma \vdash \kappa : \Box$.

- $([\Gamma], T_1) \to_I ([\Gamma], T_2)$ *if* $T_1 \equiv \forall x : T.T'$ *and* $T_2 \equiv [t/x]T'$ *for some* t *such that* $\Gamma \vdash t : T$.

- $([\Gamma, x : T], T_1) \to_G ([\Gamma], T_2)$ *if* $T_2 \equiv \forall x : T.T_1$ *and* $\Gamma \vdash T : *$.

Intuitively, $([\Gamma], T_1) \to ([\Gamma'], T_2)$ means T_1 can be transformed to T_2 with a change of context from Γ to Γ'. One can view morphing relations as a way to model typing rules which are not syntax-directed. Note that morphing relations are not intended to be viewed as rewrite relation. Instead of proving confluence for these morphing relations, we try to use substitutions to *summarize* the effects of a sequence of morphing relations. Before we do that, first we "lift" $=_{\beta,\iota}$ to a form of morphing relation.

Definition 63. $([\Gamma], T) =_{\beta,\iota} ([\Gamma], T')$ *if* $\Gamma \vdash T =_{\beta,\iota} T'$ *and* $\Gamma \vdash T : *$ *and* $\Gamma \vdash T' : *$.

The best way to understand the E, G mappings below is through understanding Lemmas 25 and 26. They give concrete demonstrations of how to *summarize* a sequence of morphing relations.

Definition 64.

$$\begin{array}{lll} E(\forall X:\kappa.T) := E(T) & E(X) := X & E(\Pi x:T_1.T_2) := \Pi x:T_1.T_2 \\ E(\lambda X.T) := \lambda X.T & E(T_1 T_2) := T_1 T_2 & E(\forall x:T'.T) := \forall x:T'.T \\ E(\iota x.T) := \iota x.T & E(T\ t) := T\ t & E(\lambda x.T) := \lambda x.T \end{array}$$

Definition 65.

$$\begin{array}{lll} G(\forall X:\kappa.T) := \forall X:\kappa.T & G(X) := X & G(\Pi x:T_1.T_2) := \Pi x:T_1.T_2 \\ G(\lambda X.T) := \lambda X.T & G(T_1 T_2) := T_1 T_2 & G(\forall x:T'.T) := G(T) \\ G(\iota x.T) := \iota x.T & G(T\ t) := T\ t & G(\lambda x.T) := \lambda x.T \end{array}$$

Lemma 24. *$E([T'/X]T) \equiv [T''/X]E(T)$ for some T''; $G([t/x]T) \equiv [t/x]G(T)$.*

Proof. By induction on the structure of T. □

Lemma 25. *If $([\Gamma], T) \to^*_{i,g} ([\Gamma'], T')$, then there exists a type substitution σ such that $\sigma E(T) \equiv E(T')$.*

Proof. It suffices to consider $([\Gamma], T) \to_{i,g} ([\Gamma'], T')$. If $T' \equiv \forall X:\kappa.T$ and $\Gamma = \Gamma', X:\kappa$, then $E(T') \equiv E(T)$. If $T \equiv \forall X:\kappa.T_1$ and $T' \equiv [T''/X]T_1$ and $\Gamma = \Gamma'$, then $E(T) \equiv E(T_1)$. By Lemma 32, we know $E(T') \equiv E([T''/X]T_1) \equiv [T_2/X]E(T_1)$ for some T_2. □

Lemma 26. *If $([\Gamma], T) \to^*_{I,G} ([\Gamma'], T')$, then there exists a term substitution δ such that $\delta G(T) \equiv G(T')$.*

Proof. It suffices to consider $([\Gamma], T) \to_{I,G} ([\Gamma'], T')$. If $T' \equiv \forall x:T_1.T$ and $\Gamma = \Gamma', x:T_1$, then $G(T') \equiv G(T)$. If $T \equiv \forall x:T_2.T_1$ and $T' \equiv [t/x]T_1$ and $\Gamma = \Gamma'$, then $E(T) \equiv E(T_1)$. By Lemma 32, we know $E(T') \equiv E([t/x]T_1) \equiv [t/x]E(T_1)$. □

Lemma 27. *If* $([\Gamma], \Pi x : T_1.T_2) \to^*_{i,g} ([\Gamma'], \Pi x : T'_1.T'_2)$*, then there exists a type substitution* σ *such that* $\sigma(\Pi x : T_1.T_2) \equiv \Pi x : T'_1.T'_2$*.*

Proof. By Lemma 25. □

Lemma 28. *If* $([\Gamma], \Pi x : T_1.T_2) \to^*_{I,G} ([\Gamma'], \Pi x : T'_1.T'_2)$*, then there exists a term substitution* δ *such that* $\delta(\Pi x : T_1.T_2) \equiv \Pi x : T'_1.T'_2$*.*

Proof. By Lemma 26. □

Let $\to^*_{\iota,\beta,i,g,I,G}$ denote $(\to_{i,g,I,G} \cup =_{\iota,\beta})^*$. Let $\to_{\iota,\beta,i,g,I,G}$ denote $\to_{i,g,I,G} \cup =_{\iota,\beta}$. The goal of confluence analysis and morph analysis is to establish the following *compatibility* theorem.

Theorem 11 (Compatibility)**.** *If* $([\Gamma], \Pi x : T_1.T_2) \to^*_{\iota,\beta,i,g,I,G} ([\Gamma'], \Pi x : T'_1.T'_2)$*, then there exists a mixed substitution*[4] ϕ *such that* $([\Gamma], \phi(\Pi x : T_1.T_2)) =_{\iota,\beta} ([\Gamma], \Pi x : T'_1.T'_2)$*. Thus* $\Gamma \vdash \phi T_1 =_\beta T'_1$ *and* $\Gamma \vdash \phi T_2 =_\beta T'_2$ *(by Theorem 10).*

Proof. By Lemma 33 and 27, making use of the fact that if $\Gamma \vdash t =_{\iota,\beta} t'$, then for any mixed substitution ϕ, we have $\Gamma \vdash \phi t =_{\iota,\beta} \phi t'$. □

Theorem 12 (Type Preservation)**.** *If* $\Gamma \vdash t : T$ *and* $\Gamma \vdash t \to_\beta t'$ *and* $\Gamma \vdash \mathsf{wf}$*, then* $\Gamma \vdash t' : T$*.*

5.6 $0 \neq 1$ in S

The proof of $0 \neq 1$ follows the same method as in Theorem 5, while emptiness of $\bot$ needs the erasure and preservation theorems. Notice that in this section, by $a = b$, we mean $\forall C : A \to *.C\ a \to C\ b$ with $a, b : A$.

[4]A substitution that contains both term substitution and type substitution.

Definition 66. $\bot := \forall A : *.\forall x : A.\forall y : A.x = y$.

Theorem 13. *There is no term t such that $\mu_c \vdash t : \bot$*

Proof. Suppose $\mu_c \vdash t : \bot$. By the erasure theorem (Theorem 8) in Section 5.5.1, we have $F(\mu_c) \vdash t : \forall A : *.\forall C : *.C \to C$ in $\mathbf{F}_\omega$. We know that $\forall A : *.\forall C : *.C \to C$ is the singleton type[5], which is inhabited by $\lambda z.z$. This means $t \to_\beta^* \lambda z.z$ (the term reductions of $\mathbf{F}_\omega$ with let-bindings are the same as $\mathbf{S}$) and $\mu_c \vdash \lambda z.z : \bot$ in $\mathbf{S}$ (by type preservation, Theorem 12). Then we would have $\mu_c, A : *, x : A, y : A, C : A \to *, z : C\ x \vdash z : C\ y$. We know this derivation is impossible since $C\ x \not\cong C\ y$.

$\square$

Theorem 14. $\mu_c \vdash 0 = 1 \to \bot$.

Proof. This proof follows the method in Theorem 5. Let $\Gamma = \mu_c, a : (\forall B : \mathsf{Nat} \to *.B\ 0 \to B\ 1), A : *, x : A, y : A, C : A \to *, c : C\ x$. We want to construct a term of type $C\ y$. Let $F := \lambda n[: \mathsf{Nat}].n\ [\lambda p : \mathsf{Nat}.A]\ (\lambda q[: A].y)x$, and note that $F : \mathsf{Nat} \to A$. We know that $F\ 0 =_\beta x$ and $F\ 1 =_\beta y$. So we can indeed convert the type of c from $C\ x$ to $C\ (F\ 0)$. And then we instantiate the B in $\forall B : \mathsf{Nat} \to *.B\ 0 \to B\ 1$ with $\lambda x[: \mathsf{Nat}].C\ (F\ x)$. So we have $C\ (F\ 0) \to C\ (F\ 1)$ as the type of a. So $a\ c : C\ (F\ 1)$, which means $a\ c : C\ y$. So we have just shown how to inhabit $0 = 1 \to \bot$ in $\mathbf{S}$. $\square$

5.7 Summary

We have revisited lambda encodings in type theory, and shown how a new self type construct $\iota x.T$ supports dependent eliminations with lambda encodings,

[5]Note that we are dealing with Curry-style $\mathbf{F}_\omega$.

including induction principles. We considered System **S**, which incorporates self types together with implicit products and a restricted version of global positive recursive definition. The corresponding induction principles for Church- and Parigot-encoded datatypes are derivable in **S**. By changing the notion of contradiction from explosion to equational inconsistency, we are able to show $0 \neq 1$ in both **CC** and **S**. We proved type preservation, which is nontrivial for **S** since several rules are not syntax-directed. We also defined an erasure from **S** to $\mathbf{F}_\omega$ with positive definitions, and proved strong normalization of **S** by showing strong normalization of $\mathbf{F}_\omega$ with positive definitions.

CHAPTER 6

LAMBDA ENCODING WITH COMPREHENSION

In this chapter, we will investigate iota-binder from a different perspective. Instead of viewing iota-binder as a type construct, we view it as a set-forming construct. For example, if $F[x]$ is a formula containing a free term variable x , then $\iota x.F[x]$ describes a set of terms t, which satisfies the formula, i.e. $t \in \iota x.F[x]$ iff $F[t]$. Recalled that in Chapter 5, we have $\vdash t : \iota x.T$ iff $\vdash t : [t/x]T$. If we compare $t \in \iota x.F[x]$ with $\vdash t : \iota x.T$, we observe that there is a similarity between the meta-level typing relation (denoted by ":") and the set membership notation "$\in$", which lies in the object logic. This observation is inspired from our earlier work on internalization ([19], see also Chapter 4.). Right now we are being informal, because it is hard to draw a connection between $F[t]$ and $\vdash t : [t/x]T$, since equating $t \in F[t]$ with $F[t]$ violates the grammatical structure of the logic. Furthermore, one can not naïvely view self type described in Chapter 5 as formula. Suppose both $\iota x.T$ and T are corresponding to formulas, we know that for the formula $F[x]$, the $\iota x.F[x]$ is representing a set, not a formula, so it is again incoherent to equates $\iota x.F[x]$ with $\iota x.T$. Nonetheless, the observation above motivates us to investigate iota-binder from a pure logical perspective.

Another source of inspiration of our work in this Chapter is from Hatcher's formulation of Frege's logic [26]. Hatcher present Frege's system [17] in modern notations, i.e. a logic with basic set-like construct and comprehension axiom. He shows how to prove all of Peano's axioms in Frege's system. Despite Frege's system is inconsistent, the development of Peano's axioms, especially the derivation of induction principle is remarkable and should be emphasis over the inconsistency.

We first present Frege's System $\mathfrak{F}$ (section 6.1), to motivate our construction of arithmetic with lambda calculus. Then we give a formulation of second order theory of lambda calculus based on iota-binder ι and epsilon relation ϵ, we call it $\mathfrak{G}$ (section 6.2). There are at least three similar systems, namely, Girard's formulation of $\mathbf{HA}_2$ à la Takeuti [24], Krivine's $\mathbf{FA}_2$ [31] and Takeuti's second order logic [48]. There are two subtle differences between $\mathfrak{G}$ and these systems, the first one is that the domain of individuals of $\mathfrak{G}$ is lambda terms instead of primitive notion of numbers. The second one is that $\mathfrak{G}$ has (ϵ, ι)-notation, namely, set-abstraction and membership relation are explict in the object language, thus comprehension axiom is needed in $\mathfrak{G}$. While the other systems use predication instead of membership relation, and set-abstraction is implicit at the meta-level, the comprehension axiom is admissible by performing substitution. We found that with explicit (ϵ, ι)-notation and explicit comprehension axiom are easier to extend to full fledge higher order system and easier to implement(see Chapter 7). In section 6.3, we define a notion of polymorphic-dependent typing within $\mathfrak{G}$, which benefits from the facts that $\mathfrak{G}$ adimits explicit (ϵ, ι)-notation. We prove all of Peano's axioms in section 6.4. We enrich the reduction on lambda term with η-and Ω-reductions, then we are able to show that the member of the inductively defined sets such as Nat is terminating with respect to *head beta-reduction* (section 6.5). Finally, we show the notion of Leibniz equality in $\mathfrak{G}$ is *faithful* to the conversions in lambda calculus (section 6.6.3).

6.1 Frege's System $\mathfrak{F}$

Certain inconsistent systems and their corresponding antinomies are invaluable, because not only the antinomies can be served as criterions for maintaining consistency, but also, perhaps more importantly, they give us examples to see how to reconstruct a large part of mathematics within these systems. Frege's system (à la Hatcher) belongs to this category[1]. In fact, $\mathfrak{G}$ is inspired by the Fregean construction of numbers. We formalize an intuitionistic version of Frege's system $\mathfrak{F}$, and then we show how to derive basic arithmetic with $\mathfrak{F}$ and how the antinomy arises.

Definition 67 (Syntax)**.**

Domain Terms/Set $a, b, s \ ::= \ x \mid \iota x.F$

Formula $F \ ::= \bot \mid s \epsilon s' \mid \ F_1 \to F_2 \mid \forall x.F \mid F \wedge F'$

Context $\Gamma \ ::= \ \cdot \mid \Gamma, F$

We identify three syntactical categories in $\mathfrak{F}$, namely, *domain terms*, *set* and *formula*. Note that *set* in this chapter is just a name for a syntactical category, we should not confused the notion of set in this chapter with the *set* in set theory like **ZF**. Note that the notion of set coincides with the notion of domain terms in $\mathfrak{F}$.

Definition 68 (Deduction Rules)**.** $\boxed{\Gamma \vdash F}$

[1]Of course, one should also mention Church's lambda calculus.

$$\frac{F \in \Gamma}{\Gamma \vdash F} \qquad \frac{\Gamma \vdash F_1 \quad F_1 \cong F_2}{\Gamma \vdash F_2} \qquad \frac{\Gamma \vdash F \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \forall x.F}$$

$$\frac{\Gamma \vdash \forall x.F}{\Gamma \vdash [s/x]F} \qquad \frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \to F_2} \qquad \frac{\Gamma \vdash F_1 \to F_2 \quad \Gamma \vdash F_1}{\Gamma \vdash F_2}$$

$$\frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_i} \qquad \frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2}$$

Note that $F_1 \cong F_2$ is specified by the comprehension axiom.

Definition 69 (Comprehension). $s\epsilon(\iota x.F) \cong [s/x]F$

Comprehension axiom is essential for Fregean number construction. The definition of number, the induction principle for numbers rely on comprehension. Because the notion domain terms and set coincide, with comprehension axiom, $\mathfrak{F}$ is inconsistent. We will show this later.

Definition 70 (Equality). $a = b := \forall z.(z\epsilon a \equiv z\epsilon b)$.

For convenient, we write $a \equiv b$ to denote $a \to b. \wedge .b \to a$. We also write $a \neq b$ for $a = b \to \bot$, $\exists a.A$ for $(\forall a.(A \to \bot)) \to \bot$. Now we can proceed to construct a naïve set theory in $\mathfrak{F}$.

Definition 71 (Naïve Set Theory).

$\Lambda := \iota x.(x = x \to \bot)$.

$\{b\} := \iota y.y = b$

$\bar{c} := \iota y.(y\epsilon c \to \bot)$.

$a \cap b := \iota z.(z\epsilon a \wedge z\epsilon b)$

$a \cup b := \iota z.(z\epsilon a \to \bot. \wedge .z\epsilon b \to \bot) \to \bot$

Theorem 15.

$\vdash \forall x.(x = x)$

$\vdash \forall x.(x \epsilon \Lambda \to \bot)$.

We can take $x\epsilon\Lambda$ as our notion of contradictory because $x\epsilon\Lambda$ implies $\bot$. We now can develop an elementary number theory in $\mathfrak{F}$.

Definition 72 (Fregean Numbers)**.**

$N := \iota x.\forall c.(\forall y.(y\epsilon c \to Sy\epsilon c)) \to 0\epsilon c \to x\epsilon c.$

$0 := \{\Lambda\}$.

$S\ a := \iota y.\exists z.(z\epsilon y. \wedge .(y \cap \overline{\{z\}})\epsilon a)$.

Theorem 16.

$\vdash 0\epsilon N$.

Proof. We want to prove $\forall c.(\forall y.(y\epsilon c \to Sy\epsilon c)) \to 0\epsilon c \to 0\epsilon c$. Assume $\forall y.(y\epsilon c \to Sy\epsilon c)$ and $0\epsilon c$, we want to show $0\epsilon c$, which is obvious[2]. □

Theorem 17. $\vdash \forall y.(y\epsilon N \to Sy\epsilon N)$.

Proof. Assume $y\epsilon N$, we want to show $Sy\epsilon N$. By comprehension, we want to show $\forall c.(\forall y.(y\epsilon c \to Sy\epsilon c)) \to 0\epsilon c \to (Sy)\epsilon c$. So we assume $\forall y.(y\epsilon c \to Sy\epsilon c)$ and $0\epsilon c$, we need to show $(Sy)\epsilon c$. We know that $y\epsilon N$ implies $\forall c.(\forall y.(y\epsilon c \to Sy\epsilon c)) \to 0\epsilon c \to y\epsilon c$. By modus ponens, we have $y\epsilon c$. By universal instantiation, we have $y\epsilon c \to Sy\epsilon c$. So by modus ponens, we have $Sy\epsilon c$. Thus we have the proof[3]. □

[2]Observe that the lambda term for the proof is Church numberal zero $\lambda s.\lambda z.z$.

[3]The lambda term for this proof is Church successor $\lambda n.\lambda s.\lambda z.s(n\ s\ z)$.

Theorem 18 (Induction Principle). $\vdash \forall c.(\forall y.(y\epsilon c \to Sy\epsilon c)) \to 0\epsilon c \to \forall x.(x\epsilon N \to x\epsilon c)$.

Proof. Assume $\forall y.(y\epsilon c \to Sy\epsilon c), 0\epsilon c, x\epsilon N$. We want to show $x\epsilon c$. We know $x\epsilon N$ implies $\forall c.(\forall y.(y\epsilon c \to Sy\epsilon c)) \to 0\epsilon c \to x\epsilon c$. By modus ponens, we get $x\epsilon c$[4]. $\square$

Observe that there is an algorithmic interpretation for constructive proof of totality of certain kind of function. For example, the proof of S is total, namely, $\forall y.(y\epsilon N \to Sy\epsilon N)$, can be encoded as Church numeral's successor $\lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$. This result is already known by Leivant and Krivine [32], [31]. So one should at least admit there is a constructive flavor in Fregean construction of number. Of course, the system itself is inconsistent, i.e. the following formula is provable in system $\mathfrak{F}$:

Let $A := (\iota u_1.u_1 \notin u_1)\epsilon(\iota u_1.u_1 \notin u_1) \cong A \to \bot$. So we have $\vdash A \to \bot$ because $A \vdash A$ and $A \vdash A \to \bot$. Also, $A \to \bot \vdash A \to \bot$ implies $A \to \bot \vdash A$, thus $\vdash (A \to \bot) \to \bot$. By modus ponens, we can derive $\vdash \bot$. It is worthnoting that intuitionistic is irrelavant to prevent inconsistency.

6.2 System $\mathfrak{G}$

System $\mathfrak{G}$ is inspired by Frege's $\mathfrak{F}$ and the possibility of understanding the iota-binder as set-abstraction in higher order logic. System $\mathfrak{G}$ is a *simple* logical system with the (ϵ, ι)-notation.

Definition 73.

Formula $F ::= X^0 \mid t\epsilon S \mid \Pi X^1.F \mid F_1 \to F_2 \mid \forall x.F \mid \Pi X^0.F$

[4]The lambda term for this proof is iterator $\lambda f.\lambda a.\lambda n.n\ f\ a$.

Set $S \ ::= X^1 \mid \iota x.F$

Domain Terms/Pure Lambda Terms $t \ ::= \ x \mid \lambda x.t \mid tt'$

Context $\Gamma \ ::= \ \cdot \mid \Gamma, F$

Note that X^0 is a formula variable, it represents any formula. X^1 is a set variable, it represents any set. $\iota x.F$ is the set formed by the formula F. To avoid inconsistency arised in $\mathfrak{F}$, we separate the notion of set and domain terms, the domain terms in $\mathfrak{G}$ are pure lambda terms. Set can only occur inside of a formula, they do not have their own rule and identity outside of a formula. Again, please do not confuse the set in this Chapter with the set in **ZF**. $\Pi X^0.F$ is a formula formed by quantifying over formula and $\Pi X^1.F$ is formed by quantifying over set. The notation of ϵ, ι are formal parts of the language of $\mathfrak{G}$, they are called (ϵ, ι)-notation.

Definition 74 (Deduction Rules). $\boxed{\Gamma \vdash F}$

$$\frac{F \in \Gamma}{\Gamma \vdash F} \qquad \frac{\Gamma \vdash F_1 \quad F_1 =_{\beta,\iota} F_2}{\Gamma \vdash F_2}\ \mathit{Conv} \qquad \frac{\Gamma \vdash F \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \forall x.F}$$

$$\frac{\Gamma \vdash \forall x.F}{\Gamma \vdash [t/x]F} \qquad \frac{\Gamma \vdash F \quad X^i \notin \mathrm{FV}(\Gamma) \quad i \in \{0,1\}}{\Gamma \vdash \Pi X^i.F} \qquad \frac{\Gamma \vdash \Pi X^0.F}{\Gamma \vdash [F'/X^0]F}\ \mathit{Inst0}$$

$$\frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \to F_2} \qquad \frac{\Gamma \vdash F_1 \to F_2 \quad \Gamma \vdash F_1}{\Gamma \vdash F_2} \qquad \frac{\Gamma \vdash \Pi X^1.F}{\Gamma \vdash [S/X^1]F}\ \mathit{Inst1}$$

The rule *Inst0* allows us to instantiate X^0 with any formula, this is what the instantiation does in system **F**, while the *Inst1* rule allows us to instantiate a set variable X^1 with any set S.

Definition 75 (Axioms). *$F_1 =_{\iota,\beta} F_2$ iff one the the following holds.*

1. *F_1 (or F_2) is of the form $t\epsilon(\iota x.F)$ and F_2 (or F_1) is of the form $[t/x]F$.*

2. *F_1 (or F_2) contains a term t and F_2 (or F_1) is obtained from F_1 by replacing t with its beta-equivalent term t'.*

The first axiom corresponds to the comprehension axiom. The second axiom corresponds to the axiom of extensionality [26], it also depends on beta-conversion axiom in lambda calculus. We know that beta-conversion in lambda calculus is Church-Rosser, thus not every lambda terms are considered equal. The reason we set up axioms through the *Conv* rule is that it will not affect the overall proof tree, a direct consequence is that the consistency and subject reduction are relatively easy to prove, as we shall see next.

6.2.1 Consistency of System $\mathfrak{G}$

We have presented the whole specifications of $\mathfrak{G}$. Now we show $\mathfrak{G}$ is consistent, in the sense that not every formula is provable in $\mathfrak{G}$. To prove consistency, we will first devise a version of $\mathfrak{G}$ with proof term annotation, denoted by $\mathfrak{G}[p]$. Then a forgetful mapping from $\mathfrak{G}[p]$ to System $\mathbf{F}$ is defined. Finally, any derivable judgement in $\mathfrak{G}[p]$ can be mapped to a deravable judgement in System $\mathbf{F}$. Thus we can conclude the proof term for $\mathfrak{G}[p]$ is strongly normalizing and not every formula in $\mathfrak{G}$ is provable.

Definition 76 (System $\mathfrak{G}[p]$)**.**

Proof Terms $p \ ::= \ a \mid \lambda a.p \mid pp'$

Proof Context $\Gamma \ ::= \ \cdot \mid a : F, \Gamma$

Definition 77 (Proof Annotation)**.** $\boxed{\Gamma \vdash p : F}$

$$\frac{\Gamma \vdash p : F \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash p : \forall x.F} \qquad \frac{\Gamma \vdash p : F_1 \quad F_1 =_{\beta,\iota} F_2}{\Gamma \vdash p : F_2}$$

$$\frac{(a : F) \in \Gamma}{\Gamma \vdash a : F} \qquad \frac{\Gamma \vdash p : \forall x.F}{\Gamma \vdash p : [t'/x]F}$$

$$\frac{\Gamma \vdash p : F \quad X^i \notin \mathrm{FV}(\Gamma) \quad i = 0, 1}{\Gamma \vdash p : \Pi X^i.F} \qquad \frac{\Gamma \vdash p : \Pi X^0.F}{\Gamma \vdash p : [F'/X^0]F}$$

$$\frac{\Gamma, a : F_1 \vdash p : F_2}{\Gamma \vdash \lambda a.p : F_1 \to F_2} \qquad \frac{\Gamma \vdash p : F_1 \to F_2 \quad \Gamma \vdash p' : F_1}{\Gamma \vdash pp' : F_2}$$

$$\frac{\Gamma \vdash p : \Pi X^1.F}{\Gamma \vdash p : [S/X^1]F}$$

The proof terms only annotated the introduction and elimination rules of implication. We say it is in Curry style.

Definition 78. *We define ϕ to be a map from $\mathfrak{G}[p]$ to System* **F**.

$\phi(X^0) := X$	$\phi(t\epsilon S) := \phi(S)$
$\phi(F_1 \to F_2) := \phi(F_1) \to \phi(F_2)$	$\phi(\Pi X^0.F) := \Pi X.\phi(F)$
$\phi(\Pi X^1.F) := \Pi X.\phi(F)$	$\phi(\forall x.F) := \phi(F)$
$\phi(X^1) := X$	$\phi(\iota x.F) := \phi(F)$

Note that the function ϕ can be easily extended to the proof context. It maps formula and set in $\mathfrak{G}[p]$ to types in System **F**.

Lemma 29.

1. *If $F_1 =_{\beta,\iota} F_2$, then $\phi(F_1) \equiv \phi(F_2)$.*

2. *$\phi(F) \equiv \phi([t'/x]F)$.*

3. *$\phi([F'/X^0]F) \equiv [\phi(F')/X]\phi(F)$.*

4. *$\phi([S/X^1]F) \equiv [\phi(S)/X]\phi(F)$.*

The following theorem connects System $\mathfrak{G}[p]$ with System **F**.

Theorem 19. *If $\Gamma \vdash p : F$ in $\mathfrak{G}[p]$, then $\phi(\Gamma) \vdash p : \phi(F)$ in $\mathbf{F}$.*

Proof. By induction on the derivation of $\Gamma \vdash p : F$.

- **Case**: $\dfrac{(a : F) \in \Gamma}{\Gamma \vdash a : F}$

  By $a : \phi(F) \in \phi(\Gamma)$.

- **Case**: $\dfrac{\Gamma \vdash p : F \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash p : \forall x.F}$

  By IH, we know that $\phi(\Gamma) \vdash p : \phi(F) \equiv \phi(\forall x.F)$.

- **Case**: $\dfrac{\Gamma \vdash p : F_1 \quad F_1 =_{\beta,\iota} F_2}{\Gamma \vdash p : F_2}$

  By lemma 29, we know that $\phi(F_1) \equiv \phi(F_2)$.

- **Case**: $\dfrac{\Gamma \vdash p : \forall x.F}{\Gamma \vdash p : [t'/x]F}$

  By lemma 29, we know that $\phi(\forall x.F) \equiv \phi(F) \equiv \phi([t'/x]F)$.

- **Case**: $\dfrac{\Gamma \vdash p : F \quad X^i \notin \mathrm{FV}(\Gamma) \quad i = 0, 1}{\Gamma \vdash p : \Pi X^i.F}$

  By IH, we know $\phi(\Gamma) \vdash p : \phi(F)$. And $X \notin \mathrm{FV}(\phi(\Gamma))$, thus $\phi(\Gamma) \vdash p : \Pi X.\phi(F) \equiv \phi(\Pi^i X.F)$.

- **Case**: $\dfrac{\Gamma \vdash p : \Pi X^0.F}{\Gamma \vdash p : [F'/X^0]F}$

  By IH, we know that $\phi(\Gamma) \vdash p : \Pi X.\phi(F)$. Thus $\phi(\Gamma) \vdash p : [\phi(F')/X]\phi(F) \equiv \phi([F'/X^0]F)$. The last equality is by lemma 29.

- **Case**: $\dfrac{\Gamma, a : F_1 \vdash p : F_2}{\Gamma \vdash \lambda a.p : F_1 \to F_2}$

  By IH, we know $\phi(\Gamma), a : \phi(F_1) \vdash p : \phi(F_2)$. Thus $\phi(\Gamma) \vdash \lambda a.p : \phi(F_1) \to \phi(F_2)$.

- **Case**: $\dfrac{\Gamma \vdash p : F_1 \to F_2 \quad \Gamma \vdash p' : F_1}{\Gamma \vdash pp' : F_2}$

  By IH, $\phi(\Gamma) \vdash p : \phi(F_1) \to \phi(F_2)$ and $\phi(\Gamma) \vdash p' : \phi(F_1)$. Thus $\phi(\Gamma) \vdash pp' : \phi(F_2)$.

- **Case**: $\dfrac{\Gamma \vdash p : \Pi X^1.F}{\Gamma \vdash p : [S/X^1]F}$

  By IH, we know that $\phi(\Gamma) \vdash p : \Pi X.\phi(F)$. Thus $\phi(\Gamma) \vdash p : [\phi(S)/X]\phi(F) \equiv \phi([S/X^1]F)$. The last equality is by lemma 29.

□

Theorem 19 implies that if $\Gamma \vdash p : F$ in $\mathfrak{G}[p]$, then p is strongly normalizing. So the formlua $\Pi X^0.X$ in $\mathfrak{G}$ is unprovable.

6.2.2 Preservation Theorem for $\mathfrak{G}[p]$

We need to establish preservation property (subject reduction) for $\mathfrak{G}[p]$ in order to explore more unprovable formulas in $\mathfrak{G}$ (at meta-level). The proof of preservation theorem is an adaption of Barendregt's method for proving preservation for System **F** à la Curry [6].

Definition 79 (Formula Reduction)**.**

- *$F_1 \to_\beta F_2$ if $t_1 =_\beta t_2$, $F_1 \equiv F[t_1]$ and $F_2 \equiv F[t_2]$.*

- *$F_1 \to_\iota F_2$ if $F_1 \equiv t\epsilon\iota x.F$ and $F_2 \equiv [t/x]F$.*

Note that $F[t_1]$ means the lambda term t_1 appears inside the formula F and $\to_{\beta,\iota}$ denotes $\to_\beta \cup \to_\iota$.

Lemma 30. *$\to_{\beta,\iota}$ is confluent.*

Proof. We know that $\to_\beta$ and $\to_\iota$ are confluent. We also know that $\to_\beta$ commutes with $\to_\iota$, so $\to_{\beta,\iota}$ is confluent. $\square$

Definition 80 (Morphing Relations)**.**

- $F_1 \to_i F_2$ *if* $F_1 \equiv \forall x.F$ *and* $F_2 \equiv [t/x]F$ *for some term* t*.*
- $F_1 \to_g F_2$ *if* $F_2 \equiv \forall x.F_1$*.*
- $F_1 \to_I F_2$ *if* $F_1 \equiv \Pi X^0.F$ *and* $F_2 \equiv [F'/X^0]F$ *for formula* F'*.*
- $F_1 \to_G F_2$ *if* $F_2 \equiv \Pi X^0.F_1$*.*
- $F_1 \to_{is} F_2$ *if* $F_1 \equiv \Pi X^1.F$ *and* $F_2 \equiv [S/X^1]F$ *for some set* S*.*
- $F_1 \to_{gs} F_2$ *if* $F_2 \equiv \Pi X^1.F_1$*.*

Let $\twoheadrightarrow_{gi}$ denotes the reflexive and transitive closure of $\to_{i,g,I,G,is,gs}$.

Lemma 31. *Suppose no free variable of* F *occurs in* Γ*. If* $\Gamma \vdash p : F$ *and* $F \twoheadrightarrow_{gi} F'$*, then* $\Gamma \vdash p : F'$*.*

Definition 81.

$$\begin{array}{lll} E_0(\Pi X^0.F) := E_0(F) & E_0(X^0) := X^0 & E_0(F_1 \to F_2) := F_1 \to F_2 \\ E_0(\Pi X^1.F) := \Pi X^1.F & E_0(\forall x.F) := \forall x.F & E_0(t \epsilon S) := t \epsilon S \end{array}$$

Definition 82.

$$\begin{array}{lll} E_1(\Pi X^1.F) := E_1(F) & E_1(X^0) := X^0 & E_1(F_1 \to F_2) := F_1 \to F_2 \\ E_1(t \epsilon S) := t \epsilon S & E_1(\forall x.F) := \forall x.F & E_1(\Pi X^0.F) := \Pi X^0.F \end{array}$$

Definition 83.

$$\begin{array}{lll} G(\Pi X^i.F) := \Pi X^i.F & G(X^0) := X^0 & G(F_1 \to F_2) := F_1 \to F_2 \\ G(\forall x.F) := G(F) & G(t \epsilon S) := t \epsilon S & \end{array}$$

Lemma 32. *$E_0([F'/X^0]F) \equiv [F''/X^0]E_0(F)$ for some F'';*

$E_1([S/X^1]F) \equiv [S/X^1]E_1(F)$; $G([t/x]F) \equiv [t/x]G(F)$.

Proof. Proof by induction on the structure of F. □

Lemma 33. *If $F \twoheadrightarrow_{i,g} F'$, then there exist a substitution δ with domain of term variables and codomain of terms such that $\delta G(F) \equiv G(F')$.*

Proof. It suffices to consider $F \rightarrow_{i,g} F'$. If $F' \equiv \forall x.F$, then $G(F') \equiv G(F)$. If $F \equiv \forall x.F_1$ and $F' \equiv [t/x]F_1$, then $G(F) \equiv G(F_1)$. By lemma 32, we know $G(F') \equiv G([t/x]F_1) \equiv [t/x]G(F_1)$. □

Lemma 34. *If $F \twoheadrightarrow_{I,G} F'$, then there exist a substitution δ with domain of formula variables and codomain of formulas such that $\delta E_0(F) \equiv E_0(F')$.*

Proof. It suffices to consider $F \rightarrow_{I,G} F'$. If $F' \equiv \Pi X^0.F$, then $E_0(F') \equiv E_0(F)$. If $F \equiv \Pi X^0.F_1$ and $F' \equiv [F''/X^0]F_1$, then $E_0(F) \equiv E_0(F_1)$. By lemma 32, we know $E_0(F') \equiv E_0([F''/X^0]F_1) \equiv [F_2/X^0]E_0(F_1)$ for some F_2.

□

Lemma 35. *If $F \twoheadrightarrow_{is,gs} F'$, then there exist a substitution δ with domain of set variables and codomain of sets such that $\delta E_1(F) \equiv E_1(F')$.*

Proof. It suffices to consider $F \rightarrow_{is,gs} F'$. If $F' \equiv \Pi X^1.F$, then $E_1(F') \equiv E_1(F)$. If $F \equiv \Pi X^1.F_1$ and $F' \equiv [S/X^1]F_1$, then $E_1(F) \equiv E_1(F_1)$. By lemma 32, we know $E_1(F') \equiv E_1([S/X^1]F_1) \equiv [S/X^1]E_1(F_1)$. □

Theorem 20 (Compatibility)**.** *If* $(F_1 \to F_2) \to^*_{\iota,\beta,i,g,I,G,is,gs} (F'_1 \to F'_2)$*, then there exists a mixed substitution*[5] δ *such that* $\delta(F_1 \to F_2) \to_\beta F'_1 \to F'_2$*. Thus* $\delta F_1 \to_\beta F'_1$ *and* $\delta F_2 \to_\beta F'_2$*.*

Proof. By lemma 33, lemma 34 and lemma 35, we have $\delta(F_1 \to F_2) \to_{\beta,\iota} F'_1 \to F'_2$ for some mix substitution δ. Since $\to_\iota$ reduction can not happen in the sequence $\delta(F_1 \to F_2) \to_{\beta,\iota} F'_1 \to F'_2$, so we have $\delta(F_1 \to F_2) \to_\beta F'_1 \to F'_2$. Thus $\delta F_1 \to_\beta F'_1$ and $\delta F_2 \to_\beta F'_2$.

□

Lemma 36 (Inversion)**.**

- *If* $\Gamma \vdash a : F$*, then exist* F_1 *such that* $F_1 \to^*_{\iota,\beta,i,g,I,G,is,gs} F$ *and* $(a : F_1) \in \Gamma$*.*
- *If* $\Gamma \vdash p_1 p_2 : F$*, then exist* F_1, F_2 *such that* $\Gamma \vdash p_1 : F_1 \to F_2$ *and* $\Gamma \vdash p_2 : F_1$ *and* $F_2 \to^*_{\iota,\beta,i,g,I,G,is,gs} F$*.*
- *If* $\Gamma \vdash \lambda a.p : F$*, then exist* F_1, F_2 *such that* $\Gamma, a : F_1 \vdash p : F_2$ *and* $F_1 \to F_2 \to^*_{\iota,\beta,i,g,I,G,is,gs} F$*.*

Lemma 37 (Substitution)**.**

1. *If* $\Gamma \vdash p : F$*, then for any mixed substitution* δ*,* $\delta\Gamma \vdash p : \delta F$*.*
2. *If* $\Gamma, a : F \vdash p : F'$ *and* $\Gamma \vdash p' : F$*, then* $\Gamma \vdash [p'/a]p : F'$*.*

Theorem 21 (Preservation)**.** *If* $\Gamma \vdash p : F$ *and* $p \to_\beta p'$*, then* $\Gamma \vdash p' : F$*.*

[5]A substitution that contains term, set and formula substitution

Proof. We list one interesting case:

$$\frac{\Gamma \vdash p_1 : F_1 \to F_2 \quad \Gamma \vdash p_2 : F_1}{\Gamma \vdash p_1 p_2 : F_2}$$

Suppose $\Gamma \vdash (\lambda a.p_1)p_2 \to_\beta [p_2/a]p_1$. We know that $\Gamma \vdash \lambda a.p_1 : F_1 \to F_2$ and $\Gamma \vdash p_2 : F_1$. By inversion on $\Gamma \vdash \lambda a.p_1 : F_1 \to F_2$, we know that there exist F_1', F_2' such that $\Gamma, a : F_1' \vdash p_1 : F_2'$ and $(F_1' \to F_2') \to^*_{\iota,\beta,i,g,I,G,is,gs} (F_1 \to F_2)$. By theorem 20, we have $\delta(F_1' \to F_2') =_\beta (F_1 \to F_2)$. By Church-Rosser of $=_\beta$, we have $\delta F_1' =_\beta F_1$ and $\delta F_2' =_\beta F_2$. So by (1) of lemma 37, we have $\Gamma, a : \delta F_1' \vdash p_1 : \delta F_2'$. So $\Gamma, a : \delta F_1' \vdash p_1 : F_2$. Since $\Gamma \vdash p_2 : \delta F_1'$, by (2) of lemma 37, $\Gamma \vdash [p_2/a]p_1 : F_2$.

□

6.3 A Polymorphic Dependent Type System $\mathfrak{G}[t]$

In this section, we first show a polymorphic dependent type system $\mathfrak{G}[t]$. Then, we define an embedding from $\mathfrak{G}[t]$ to $\mathfrak{G}$. The embedding is invertable, thus we can transform (at meta level) a judgement in $\mathfrak{G}$ to a judgement in $\mathfrak{G}[t]$ and vice versa. We call this behavior *reciprocity.*

Definition 84 (Syntax)**.**

Lambda Terms $t \ := \ x \mid \lambda x.t \mid tt'$

Internal Types $U \ := X^1 \mid \iota x.Q \mid \Pi x : U.U' \mid \Delta X^1.U$

Internal Formula $Q \ := X^0 \mid t \epsilon U \mid \Pi X^0.Q \mid Q \to Q' \mid \forall x.Q \mid \Pi X^1.Q$

Internal Context $\Psi \ := \ \cdot \mid \Psi, x \epsilon U$

Besides basic set formed by formula and set variable, internal types includes dependent-type-like construct $\Pi x : U.U'$ and polymorphic-type-like construct $\Delta X^1.U$.

The internal formula stay the same as formula in $\mathfrak{G}$ except replacing the notion of set by the notion of internal type. We can view ϵ relation as a kind of typing relation, thus we have the notion of internal context as a list of formula of the form $x\epsilon U$ and the following internal typing relation.

Definition 85 (Internal Typing). $\boxed{\Psi \Vdash t\epsilon U}$

$$\frac{x\epsilon U \in \Psi}{\Psi \Vdash x\epsilon U} \qquad \frac{\Psi, x\epsilon U \Vdash t\epsilon U'}{\Psi \Vdash \lambda x.t\epsilon \Pi x : U.U'} \qquad \frac{\Psi \Vdash t\epsilon U \quad X^1 \notin FV(\Psi)}{\Psi \Vdash t\epsilon \Delta X^1.U}$$

$$\frac{\Psi \Vdash t\epsilon \Delta X^1.U}{\Psi \Vdash t\epsilon [U'/X]U} \quad \frac{\Psi \Vdash t_1\epsilon \Pi x : U'.U \quad \Psi \Vdash t_2\epsilon U'}{\Psi \Vdash t_1 t_2 \epsilon [t_2/x]U}$$

The internal typing looks remarkably like the usual polymorphic dependent type system. But we want to emphasis that the meaning of internal typing in $\mathfrak{G}[t]$ is different from the usual notion of typing. The internal typing relation is an internal formula in $\mathfrak{G}[t]$ (which lies in the object language), while the ususal notion of typing relation is a meta-level relation. For example, $t\epsilon U$ is a formula while $t : T$ is a meta-level relation. The emergence of internal typing benefits from the (ϵ, ι)-notation. Now let us relate $\mathfrak{G}[t]$ with $\mathfrak{G}$.

Definition 86. $[\![\cdot]\!]$ *is an* ***embedding*** *from internal types in* $\mathfrak{G}[t]$ *to sets in* $\mathfrak{G}$*, internal formulas in* $\mathfrak{G}[t]$ *to formulas in* $\mathfrak{G}$*.*

$[\![X^1]\!] := X^1$

$[\![\iota x.Q]\!] := \iota x.[\![Q]\!]$

$[\![\Pi x : U'.U]\!] := \iota f.\forall x.(x\epsilon[\![U']\!] \to f\ x\epsilon[\![U]\!])$*, where* f *is fresh.*

$[\![\Delta X^1.U]\!] := \iota x.(\Pi X^1.x\epsilon[\![U]\!])$*, where* x *is fresh.*

$[\![X^0]\!] := X^0$

$[\![t\epsilon U]\!] := t\epsilon[\![U]\!]$

$[\![Q \to Q']\!] := [\![Q]\!] \to [\![Q]\!]$

$[\![\Pi X^i.Q]\!] := \Pi X^i.[\![Q]\!].$

$[\![\forall x.Q]\!] := \forall x.[\![Q]\!].$

$[\![x\epsilon U, \Psi]\!] := x\epsilon[\![U]\!], [\![\Psi]\!]$

Lemma 38. *$[t'/x][\![U]\!] = [\![[t'/x]U]\!]$ and $[[\![U']\!]/X^1][\![U]\!] = [\![[U'/X^1]U]\!]$.*

Proof. By induction on structure of U. □

Theorem 22. *If $\Psi \Vdash t\epsilon U$, then $[\![\Psi]\!] \vdash t\epsilon[\![U]\!]$.*

Proof. By induction on the derivation of $\Psi \Vdash t\epsilon U$.

- **Case**: $\dfrac{x\epsilon U \in \Psi}{\Psi \Vdash x\epsilon U}$

  $[\![\Psi]\!] \vdash x\epsilon[\![U]\!]$, since $x\epsilon[\![U]\!] \in [\![\Psi]\!]$.

- **Case**: $\dfrac{\Psi, x\epsilon U \Vdash t\epsilon U'}{\Psi \Vdash \lambda x.t\epsilon\Pi x : U.U'}$

  By induction, we have $[\![\Psi]\!], x\epsilon[\![U]\!] \vdash t\epsilon[\![U']\!]$. So $[\![\Psi]\!] \vdash x\epsilon[\![U]\!] \to t\epsilon[\![U']\!]$, then by $\forall$-intro rule, we have $[\![\Psi]\!] \vdash \forall x.(x\epsilon[\![U]\!] \to t\epsilon[\![U']\!])$. By comprehension rule and beta-reduction, we get $[\![\Psi]\!] \vdash \lambda x.t\epsilon\iota f.\forall x.(x\epsilon[\![U]\!] \to f\ x\epsilon[\![U']\!])$. We know that $[\![\Pi x : U.U']\!] := \iota f.\forall x.(x\epsilon[\![U]\!] \to f\ x\epsilon[\![U']\!])$.

- **Case**: $\dfrac{\Psi \Vdash t\epsilon\Pi x : U'.U \quad \Psi \Vdash t'\epsilon U'}{\Psi \Vdash tt'\epsilon[t'/x]U}$

  By induction, we have $[\![\Psi]\!] \vdash t\epsilon\iota f.\forall x.(x\epsilon[\![U']\!] \to f\ x\epsilon[\![U]\!])$ and $[\![\Psi]\!] \vdash t'\epsilon[\![U']\!]$. By comprehension, we have $[\![\Psi]\!] \vdash \forall x.(x\epsilon[\![U']\!] \to t\ x\epsilon[\![U]\!])$. Instantiate x with t', we have $[\![\Psi]\!] \vdash t'\epsilon[\![U']\!] \to t\ t'\epsilon[t'/x][\![U]\!]$. So by modus ponens, we have

$[\![\Psi]\!] \vdash tt'\epsilon[t'/x][\![U]\!]$. By lemma 38, we know that $[t'/x][\![U]\!] = [\![[t'/x]U]\!]$. So $[\![\Psi]\!] \vdash tt'\epsilon[\![[t'/x]U]\!]$.

- **Case**: $\dfrac{\Psi \Vdash t\epsilon U \quad X^1 \notin FV(\Psi)}{\Psi \Vdash t\epsilon\Delta X^1.U}$

  By induction, one has $[\![\Psi]\!] \vdash t\epsilon[\![U]\!]$. So one has $[\![\Psi]\!] \vdash \Pi X^1.t\epsilon[\![U]\!]$. So by comprehension, one has $[\![\Psi]\!] \vdash t\epsilon\iota x.\Pi X^1.x\epsilon[\![U]\!]$.

- **Case**: $\dfrac{\Psi \Vdash t\epsilon\Delta X^1.U}{\Psi \Vdash t\epsilon[U'/X]U}$

  By induction, one has $[\![\Psi]\!] \vdash t\epsilon\iota x.\Pi X^1.x\epsilon[\![U]\!]$. By comprehension, we have $[\![\Psi]\!] \vdash \Pi X^1.t\epsilon[\![U]\!]$. So by instantiation, we have $[\![\Psi]\!] \vdash t\epsilon[[\![U']\!]/X^1][\![U]\!]$. Since by lemma 38, we know $[[\![U']\!]/X^1][\![U]\!] = [\![[U'/X^1]U]\!]$.

□

Definition 87.

$[\![\cdot]\!]^{-1}$ *is a maping from the sets in* $\mathfrak{G}$ *to the internal types in* $\mathfrak{G}[t]$*, from the formulas in* $\mathfrak{G}$ *to the internal formulas* $\mathfrak{G}[t]$*.*

$[\![X^1]\!]^{-1} := X^1$

$[\![\iota f.\forall x.(x\epsilon S' \to f\ x\epsilon S)]\!]^{-1} := \Pi x : [\![S']\!]^{-1}.[\![S]\!]^{-1}$*, where* f *is fresh.*

$[\![\iota x.(\Pi X^1.x\epsilon S)]\!]^{-1} := \Delta X^1.[\![S]\!]^{-1}$*, where* x *is fresh.*

$[\![\iota x.T]\!]^{-1} := \iota x.[\![T]\!]^{-1}$

$[\![X^0]\!]^{-1} := X^0$

$[\![t\epsilon S]\!]^{-1} := t\epsilon[\![S]\!]^{-1}$

$[\![T \to T']\!]^{-1} := [\![T]\!]^{-1} \to [\![T]\!]^{-1}$

$[\![\Pi X^i.T]\!]^{-1} := \Pi X^i.[\![T]\!]^{-1}$.

$[\![\forall x.T]\!]^{-1} := \forall x.[\![T]\!]^{-1}.$

$[\![x\epsilon S, \Gamma]\!]^{-1} := x\epsilon[\![S]\!]^{-1}, [\![\Gamma]\!]^{-1}$

Lemma 39. *$[\![[\![S]\!]^{-1}]\!] = S$ and $[\![[\![U]\!]]\!]^{-1} = U$.*

Proof. By induction. □

By lemma 39, if we have $\Gamma \vdash t\epsilon S$ in $\mathfrak{G}$, we can go to $\mathfrak{G}[t]$ by $[\![\Gamma]\!]^{-1} \Vdash t\epsilon[\![S]\!]^{-1}$. Then, after a few deductions in $\mathfrak{G}[t]$, we can use theorem 22 to go back to $\mathfrak{G}$.

6.4 Proving Peano's Axioms

In this section, we prove all of Peano's axioms [41] in $\mathfrak{G}$. First, let us define natural number as Scott numeral.

Definition 88 (Scott Numerals)**.**

$\mathsf{Nat} := \iota x.\Pi C^1.(\forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to x\epsilon C$

$\mathsf{S} \ := \lambda n.\lambda s.\lambda z.s\ n$

$0 \ := \lambda s.\lambda z.z$

Theorem 23 (Peano's Axiom 1)**.** $\vdash 0\epsilon\mathsf{Nat}$.

Proof. By comprehension, we want to show $\vdash \Pi C^1.(\forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to 0\epsilon C$, which is obvious[6]. □

Definition 89 (Leibniz Equality)**.** *$x = y \ := \ \Pi C^1.x\epsilon C \to y\epsilon C$.*

Theorem 24 (Peano Axiom 2-4)**.**

[6]Note the proof terms for the theorem is Church numeral 0.

1. $\forall x.x = x$.

2. $\forall x.\forall y.x = y \to y = x$.

3. $\forall x.\forall y.\forall z.x = y \to y = z \to x = z$.

Proof. We only prove 2, the others are easy. Assume $\Pi C^1.x\epsilon C \to y\epsilon C(1)$, we want to show $y\epsilon A \to x\epsilon A$ for any A^1. Instantiate C in (1) with $\iota z.(z\epsilon A \to x\epsilon A)$. By comprehension, we get $(x\epsilon A \to x\epsilon A) \to (y\epsilon A \to x\epsilon A)$. And we know that $x\epsilon A \to x\epsilon A$ is derivable in our system, so by modus ponens we get $y\epsilon A \to x\epsilon A$. □

Lemma 40. $\cdot \vdash \forall a.\forall b.\Pi P^1.(a\epsilon P \to a = b \to b\epsilon P)$.

Proof. By modus ponens. □

Theorem 25 (Peano's Axiom 5)**.** $\cdot \vdash \forall a.\forall b.(a\epsilon\mathsf{Nat} \to a = b \to b\epsilon\mathsf{Nat})$.

Proof. Let $P := \iota x.x\epsilon\mathsf{Nat}$ for lemma 40. □

Theorem 26 (Peano's Axiom 6)**.** $\cdot \vdash \forall m.(m\epsilon\mathsf{Nat} \to \mathsf{S}m\epsilon\mathsf{Nat})$.

Proof. Assume $m\epsilon\mathsf{Nat}$. We want to show $\mathsf{S}m\epsilon\mathsf{Nat}$. By comprehension, we just need to show $\Pi C^1.(\forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to \mathsf{S}m\epsilon C$. By Intros, we want to derive $m\epsilon\mathsf{Nat}, \forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C), 0\epsilon C \vdash \mathsf{S}m\epsilon C$. Since $m\epsilon\mathsf{Nat}$, we know that $\Pi C^1.(\forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to m\epsilon C$. By Modus Ponens, we have $m\epsilon C$. We know that $m\epsilon\mathsf{Nat}, \forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C), 0\epsilon C \vdash (m\epsilon C) \to (\mathsf{S}m)\epsilon C$. Thus we derive $m\epsilon\mathsf{Nat}, \forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C), 0\epsilon C \vdash \mathsf{S}m\epsilon C$, which is what we want[7]. □

[7]Note that the proof term for this theorem is Church successor.

Theorem 27 (Induction Principle)**.**

$\vdash \Pi C^1.(\forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to \forall m.(m\epsilon\mathsf{Nat} \to m\epsilon C)$

Proof. Assume $\forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C), 0\epsilon C$ and $m\epsilon\mathsf{Nat}$. We need to show that $m\epsilon C$. Since $m\epsilon\mathsf{Nat}$ implies that $\Pi C^1.(\forall y.((y\epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to m\epsilon C$. So by instantiation and modus ponens we get $m\epsilon C$. [8]

□

In order to proceed to prove Peano's axiom 7, we need to define a notion of contradiction in $\mathfrak{G}$.

Definition 90 (Notion of Contradiction)**.** $\bot := \forall x.\forall y.(x = y)$.

Theorem 28 (Consistency (Meta)[9])**.** $\bot$ *is uninhabited in* $\mathfrak{G}[p]$.

Proof. Suppose $\bot$ is inhabited, that is, there is a proof term p such that $\cdot \vdash p : \forall x.\forall y.\Pi C^1.x\epsilon C \to y\epsilon C$. By theorem 19 and theorem 21, we know that p must normalized at some normal proof term p' such that $\cdot \vdash p' : \forall x.\forall y.\Pi C^1.x\epsilon C \to y\epsilon C$. We know that p' must of the form $\lambda a.p''$ with $a : x\epsilon C$. Since $=_{\beta,\iota}$ is Church-Rosser, we can not convert $x\epsilon C$ to $y\epsilon C$. So p' can not exist. □

Lemma 41. $\vdash 0 = \mathsf{S}0 \to \bot$.

Proof. Assume $0 = \mathsf{S}0$, namely, $\Pi C^1.0\epsilon C \to \mathsf{S}0\epsilon C$ †. We want to show $\forall x.\forall y.\Pi A^1.x\epsilon A \to y\epsilon A$. Assume $x\epsilon A$ (1). We now instantiate C with $\iota u.(((\lambda n.n\ (\lambda z.y)\ x)\ u)\epsilon A)$ in †.

[8]The proof terms for this theorem is $\lambda s.\lambda z.\lambda n.n\ s\ z$.

[9]Meaning the proof of this theorem relies on meta-level argument.

By comprehension and beta reduction, we get $x\epsilon A \to y\epsilon A$ (2). By modus ponens of (1), (2), we get $y\epsilon A$. $\square$

We also need predecessor to prove Peano's axiom 7.

Definition 91. $\mathsf{Pred} := \lambda n.n(\lambda x.x)0$.

Lemma 42 (Congruence of Equality). $\vdash \forall a.\forall b.\forall f.a = b \to fa = fb.$

Proof. Assume $\Pi C.a\epsilon C \to b\epsilon C$. Let $C := \iota x.fx\epsilon P$ with P free. Instantiate C for the assumption, we get $a\epsilon(\iota x.fx\epsilon P) \to b\epsilon(\iota x.fx\epsilon P)$. By conversion, we get $f\ a\epsilon P \to f\ b\epsilon P$. So by polymorphic generalization, we get $f\ a = f\ b$.

$\square$

Theorem 29 (Peano's Axiom 7). $\vdash \forall n.n\epsilon\mathsf{Nat} \to (\mathsf{S}n = 0 \to \bot)$

Proof. We will use induction principle (theorem 27) to prove this. We instantiate C in theorem 27 with $\iota z.(\mathsf{S}z = 0 \to \bot)$, we have $\forall y.((\mathsf{S}y = 0 \to \bot) \to (\mathsf{SS}y = 0 \to \bot)) \to (\mathsf{S}0 = 0 \to \bot) \to \forall m.(m\epsilon\mathsf{Nat} \to (\mathsf{S}m = 0 \to \bot))$. Base case is by lemma 41. For the step case, we assume $\mathsf{S}y = 0 \to \bot$ (IH), we want to show $\mathsf{SS}y = 0 \to \bot$. Assuming $\mathsf{SS}y = 0$, we want to show $\bot$. By lemma 42, we know that $\mathsf{Pred}(\mathsf{SS}y) = \mathsf{Pred}0$. By beta-reduction, we have $\mathsf{S}y = 0$. Thus by IH, we have $\bot$. $\square$

Theorem 30 (Peano's Axiom 8). $\forall m.\forall n.m\epsilon\mathsf{Nat} \to n\epsilon\mathsf{Nat} \to \mathsf{S}m = \mathsf{S}n \to m = n.$

Proof. Assume $\mathsf{S}m = \mathsf{S}n$. By lemma 42, we have $\mathsf{Pred}(\mathsf{S}m) = \mathsf{Pred}(\mathsf{S}n)$. So by beta reduction, we have $m = n$. $\square$

In order to state Peano's axiom 9, we extend the formula in $\mathfrak{G}$ with $F \wedge F'$. And the proof of $F \wedge F'$ consist of both the proof of F and the proof of F'[10].

Theorem 31 (Peano's Axiom 9, Weak Induction)**.**

$\vdash \Pi C^1.(\forall y.(y\epsilon\mathsf{Nat} \wedge (y\epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to \forall m.(m\epsilon\mathsf{Nat} \to m\epsilon C)$

Proof. Assume $\forall y.(y\epsilon\mathsf{Nat} \wedge (y\epsilon C) \to (\mathsf{S}y)\epsilon C)$ † and $0\epsilon C$. We want to show that $\forall m.(m\epsilon\mathsf{Nat} \to m\epsilon C)$. We just need to show $\forall m.(m\epsilon\mathsf{Nat} \to (m\epsilon\mathsf{Nat} \wedge m\epsilon C))$. We prove this using theorem 27. For the base case, it is obvious that $0\epsilon\mathsf{Nat} \wedge 0\epsilon C$. For step case, assuming $z\epsilon\mathsf{Nat} \wedge z\epsilon C$ (IH), we need to show $\mathsf{S}z\epsilon\mathsf{Nat} \wedge \mathsf{S}z\epsilon C$. By theorem 26, we have $\mathsf{S}z\epsilon\mathsf{Nat}$. By †, we know that $\mathsf{S}z\epsilon C$. Thus $\forall z.(z\epsilon\mathsf{Nat} \to (z\epsilon\mathsf{Nat} \wedge z\epsilon C))$. □

We have proved all Peano's nine axioms. We will leave the investigation of the relation between strong induction principle and the weak induction principle as future work.

6.5 Reasoning about Programs

System $\mathfrak{G}$ is expressive enough to reason about programs. By programs we mean lambda calculus with Scott encoding and recursive term definitions. We first show some simple examples about Scott numerals, and then we show how to encode Vector in $\mathfrak{G}$.

Definition 92. $\mathsf{add} := \lambda n.\lambda m.n\ (\lambda p.\mathsf{add}\ p\ (\mathsf{S}m))\ m$

We know that the above recursive equation can be solved by fixpoint. For convenient, we simply use the definition as a kind of build-in beta equality. i.e. whenever we see a add, we one step unfold it.

[10]This extension can be avoided by defining $F \wedge F' := \forall Y^0.(F \to F' \to Y) \to Y$.

Theorem 32. $\cdot \vdash \forall n.(n\epsilon\mathsf{Nat} \to \mathsf{add}\ n\ 0 = n)$.

Proof. We want to show $\forall n.(n\epsilon\mathsf{Nat} \to \mathsf{add}\ n\ 0 = n)$. Let $P := \iota x.\mathsf{add}\ x\ 0 = x$. Instantiate the C^1 in theorem 27 with P, we get $\forall y.(\mathsf{add}\ y\ 0 = y \to \mathsf{add}\ (\mathsf{S}y)\ 0 = \mathsf{S}y) \to \mathsf{add}\ 0\ 0 = 0 \to \forall m.(m\epsilon\mathsf{Nat} \to m\epsilon P)$. We just have to prove $\forall y.(\mathsf{add}\ y\ 0 = y \to \mathsf{add}\ (\mathsf{S}y)\ 0 = \mathsf{S}y)$ and $\mathsf{add}\ 0\ 0 = 0$. For the base case, we want to show $\Pi C.\mathsf{add}\ 0\ 0\epsilon C \to 0\epsilon C$. Assume $\mathsf{add}\ 0\ 0\epsilon C$, since $\mathsf{add}\ 0\ 0 \to_\beta 0$, by conversion, we get $0\epsilon C$. For the step case is a bit complicated, assume $\mathsf{add}\ y\ 0 = y$, we want to show $\mathsf{add}\ (\mathsf{S}y)\ 0 = \mathsf{S}y$. Since $\mathsf{add}\ y\ 0 \to_\beta y\ (\lambda p.\mathsf{add}\ p\ (\mathsf{S}0))\ 0$, And $\mathsf{add}\ (\mathsf{S}y)\ 0 \to_\beta \mathsf{add}\ y\ (\mathsf{S}0) \leftarrow_\beta^* \mathsf{S}(\mathsf{add}\ y\ 0)$. So lemma 42 will give us this. □

Theorem 33. $\cdot \vdash \forall n.(n\epsilon\mathsf{Nat} \to \forall m.(m\epsilon\mathsf{Nat} \to \mathsf{add}\ n\ m\epsilon\mathsf{Nat}))$. *After transformed to* $\mathfrak{G}[t]$*, we have* $\Vdash \mathsf{add}\epsilon\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$. [11]

Proof. Let $P := \iota z.\forall m.(m\epsilon\mathsf{Nat} \to \mathsf{add}\ z\ m\epsilon\mathsf{Nat})$. We instantiate the C in theorem 27, we have $(\forall y.((y\epsilon P) \to (\mathsf{S}y)\epsilon P)) \to 0\epsilon P \to \forall m.(m\epsilon\mathsf{Nat} \to m\epsilon P)$. For the base case, we need to show $\forall m.(m\epsilon\mathsf{Nat} \to \mathsf{add}\ 0\ m\epsilon\mathsf{Nat})$. By $\mathsf{add}\ 0\ m \to_\beta m$, we have the base case. For the step case, assuming $\forall m.(m\epsilon\mathsf{Nat} \to \mathsf{add}\ y\ m\epsilon\mathsf{Nat})$ (IH), we need to show $\forall m.(m\epsilon\mathsf{Nat} \to \mathsf{add}\ (\mathsf{S}y)\ m\epsilon\mathsf{Nat})$. We know that $\mathsf{add}\ (\mathsf{S}y)\ m \to_\beta^* \mathsf{add}\ y\ (\mathsf{S}m)$. By (IH), we know $\mathsf{add}\ y\ (\mathsf{S}m)\epsilon\mathsf{Nat}$. So $\mathsf{add}\ (\mathsf{S}y)\ m\epsilon\mathsf{Nat}$. □

In order to do vector encoding in $\mathfrak{G}$, we need to extend the formulas of $\mathfrak{G}$ to specify binary relation, so we add the following syntatic category.

[11] We write $U \to U'$ if $\Pi x : U.U'$ with $x \notin \mathrm{FV}(U')$.

Definition 93 (Relation[12]).

Formula F ::= ... $\mid (t;t')\epsilon R \mid \Pi X^2.F$

Binary Relation R ::= $X^2 \mid \iota(x;y).F$

Relational Comprehension $(t;t')\epsilon\iota(x;y).F =_\iota [(t;t')/(x;y)]F$

Definition 94 (Vector).

$\mathsf{vec}(U,n) :=$

$$\iota x.\Pi C^2.(\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon C \to (\mathsf{cons}\ m\ u\ y;\mathsf{S}m)\epsilon C)) \to$$

$(\mathsf{nil};0)\epsilon C \to (x;n)\epsilon C$

$\mathsf{nil} := \lambda y.\lambda x.x$

$\mathsf{cons} := \lambda n.\lambda v.\lambda l.\lambda y.\lambda x.y\ n\ v\ l.$

Lemma 43. $\vdash \mathsf{nil}\epsilon\mathsf{vec}(U,0)$.

Lemma 44. $\vdash \forall n.n\epsilon\mathsf{Nat} \to \forall u.(u\epsilon U \to \forall l.(l\epsilon\mathsf{vec}(U,n) \to (\mathsf{cons}\ n\ u\ l)\epsilon\mathsf{vec}(U,\mathsf{S}n)))$. *Transform to* $\mathfrak{G}[t]$*, we get* $\Vdash \mathsf{cons}\epsilon\Pi n : \mathsf{Nat}.U \to \mathsf{vec}(U,n) \to \mathsf{vec}(U,\mathsf{S}n)$.

Proof. Assume $n\epsilon\mathsf{Nat}, u\epsilon U, l\epsilon\mathsf{vec}(U,n)$. We want to show $(\mathsf{cons}\ n\ u\ l)\epsilon\mathsf{vec}(U,\mathsf{S}n)$. By comprehension, we need to show $\Pi C^2.(\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon C \to (\mathsf{cons}\ m\ u\ y;\mathsf{S}m)\epsilon C)) \to (\mathsf{nil};0)\epsilon C \to ((\mathsf{cons}\ n\ u\ l);\mathsf{S}n)\epsilon C$. Assume that we have $\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon C \to (\mathsf{cons}\ m\ u\ y;\mathsf{S}m)\epsilon C)$ † and $(\mathsf{nil};0)\epsilon C$, we need to show that $((\mathsf{cons}\ n\ u\ l);\mathsf{S}n)\epsilon C$. We know that $l\epsilon\mathsf{vec}(U,n)$, by comprehension, we have

[12]We will show a more uniform extension of $\mathfrak{G}$ in next Chapter.

$$\Pi C^2.(\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon C \to (\mathsf{cons}\ m\ u\ y;\mathsf{S}m)\epsilon C)) \to (\mathsf{nil};0)\epsilon C \to (l;n)\epsilon C.$$

By modus ponens, we have $(l;n)\epsilon C$. Instantiate y with l, m with n, u with u in †, we have $n\epsilon\mathsf{Nat} \to u\epsilon U \to (l;n)\epsilon C \to (\mathsf{cons}\ n\ u\ l;\mathsf{S}n)\epsilon C$. So by modus ponens, we have $(\mathsf{cons}\ n\ u\ l;\mathsf{S}n)\epsilon C$.

□

Theorem 34 (Induction Principle)**.**

$\vdash \mathsf{Ind}(U,n) :=$

$$\Pi C^2.(\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon C \to (\mathsf{cons}\ m\ u\ y;\mathsf{S}m)\epsilon C)) \to (\mathsf{nil};0)\epsilon C \to \forall l.(l\epsilon\mathsf{vec}(U,n) \to (l;n)\epsilon C)$$

Proof. Assume we have $l\epsilon\mathsf{vec}(U,n)$ and

$$\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon C \to (\mathsf{cons}\ m\ u\ y;\mathsf{S}m)\epsilon C), (\mathsf{nil};0)\epsilon C.$$

We want to show $(l;n)\epsilon C$. By comprehension, we have

$$\Pi C^2.(\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon C \to (\mathsf{cons}\ m\ u\ y;\mathsf{S}m)\epsilon C)) \to (\mathsf{nil};0)\epsilon C \to (l;n)\epsilon C.$$

By modus ponens, we have $(l;n)\epsilon C$.

□

Definition 95 (Append)**.**

$\mathsf{app} := \lambda n_1.\lambda n_2.\lambda l_1.\lambda l_2.l_1(\lambda m.\lambda h.\lambda t.\mathsf{cons}\ (m+n_2)\ h\ (\mathsf{app}\ m\ n_2\ t\ l_2))l_2$

Theorem 35. $\Vdash \mathsf{app}\epsilon\Pi n_1 : \mathsf{Nat}.\Pi n_2 : \mathsf{Nat}.\mathsf{vec}(U,n_1) \to \mathsf{vec}(U,n_2) \to \mathsf{vec}(U,n_1+n_2)$

Proof. Note that we state the theorem in $\mathfrak{G}[t]$. So we want to derive

$n_1 \epsilon \mathsf{Nat}, n_2 \epsilon \mathsf{Nat} \Vdash \lambda l_1.\lambda l_2.l_1(\lambda m.\lambda h.\lambda t.\mathsf{cons}\ (m+n_2)\ h\ (\mathsf{app}\ m\ n_2\ t\ l_2))l_2\ \epsilon$

$\mathsf{vec}(U, n_1) \to \mathsf{vec}(U, n_2) \to \mathsf{vec}(U, n_1 + n_2)$.

We now transform it back to $\mathfrak{G}$, we have:

$n_1 \epsilon \mathsf{Nat}, n_2 \epsilon \mathsf{Nat} \vdash \forall x_1.x_1 \epsilon \mathsf{vec}(U, n_1) \to \forall x_2.(x_2 \epsilon \mathsf{vec}(U, n_2) \to x_1(\lambda m.\lambda h.\lambda t.\mathsf{cons}\ (m + n_2)\ h\ (\mathsf{app}\ m\ n_2\ t\ x_2))x_2 \epsilon \mathsf{vec}(U, n_1 + n_2))$.

We instantiate the C in theorem 34 by

$P := \iota\ (l; n)\ .\forall x_2.(x_2 \epsilon \mathsf{vec}(U, n_2) \to$

$l\ (\lambda m'.\lambda h.\lambda t.\mathsf{cons}\ (m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2 \epsilon \mathsf{vec}(U,\ n\ + n_2))$.

So we get

$(\forall y.\forall m.\forall u.(m \epsilon \mathsf{Nat} \to u \epsilon U \to (y; m) \epsilon P \to (\mathsf{cons}\, m\, u\, y; \mathsf{S} m) \epsilon P)) \to (\mathsf{nil}; 0) \epsilon P \to \forall l.(l \epsilon \mathsf{vec}(U, n) \to (l; n) \epsilon P)$.

For the base case, we can easily prove $\forall x_2.(x_2 \epsilon \mathsf{vec}(U, n_2) \to (\mathsf{nil}(\lambda m'.\lambda h.\lambda t.\mathsf{cons}\ (m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2 \epsilon \mathsf{vec}(U, 0 + n_2)))$. For the step case, assume (IH) $\forall x_2.(x_2 \epsilon \mathsf{vec}(U, n_2) \to y(\lambda m'.\lambda h.\lambda t.\mathsf{cons}\ (m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2 \epsilon \mathsf{vec}(U, m + n_2))$, we want to show that $\forall x_2.(x_2 \epsilon \mathsf{vec}(U, n_2) \to (\mathsf{cons}\ m\ u\ y)(\lambda m'.\lambda h.\lambda t.\mathsf{cons}\ (m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2 \epsilon \mathsf{vec}(U, \mathsf{S} m + n_2))$. We know that

$(\mathsf{cons}\ m\ u\ y)(\lambda m'.\lambda h.\lambda t.\mathsf{cons}\ (m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2 \to_\beta^*$

$\mathsf{cons}(m + n_2)\ u\ (\mathsf{app}\ m\ n_2\ y\ x_2) \to_\beta^*$

$\mathsf{cons}\ (m + n_2)\ u\ (y\ (\lambda m'.\lambda h.\lambda t.\mathsf{cons}(m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2))$.

By (IH), we know that

$y(\lambda m'.\lambda h.\lambda t.\mathsf{cons}\ (m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2 \epsilon \mathsf{vec}(U, m + n_2)$. By lemma 44 $\mathsf{cons}\ (m + n_2)\ u\ (y\ (\lambda m'.\lambda h.\lambda t.\mathsf{cons}(m' + n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2)) \epsilon \mathsf{vec}(U, \mathsf{S}(m + n_2))$.

Thus $(\mathsf{cons}\ m\ u\ y)(\lambda m'.\lambda h.\lambda t.\mathsf{cons}\ (m'+n_2)\ h\ (\mathsf{app}\ m'\ n_2\ t\ x_2))x_2\epsilon\mathsf{vec}(U, \mathsf{S}(m+n_2))$. Of course, we assume we have $\mathsf{S}(m+n_2) = \mathsf{S}m + n_2$, so we have the proof.

$\square$

Theorem 36 (Associativity). $\vdash \forall(n_1.n_2.n_3.v_1.v_2.v_3).(n_1\epsilon\mathsf{Nat} \to n_2\epsilon\mathsf{Nat} \to n_3\epsilon\mathsf{Nat} \to v_1\epsilon\mathsf{vec}(U, n_1) \to v_2\epsilon\mathsf{vec}(U, n_2)) \to v_3\epsilon\mathsf{vec}(U, n_3) \to$

$\mathsf{app}\ n_1\ (n_2+n_3)\ v_1\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (n_1+n_2)\ n_3\ (\mathsf{app}\ n_1\ n_2\ v_1\ v_2)\ v_3$

Proof. Assume $n_1\epsilon\mathsf{Nat}, n_2\epsilon\mathsf{Nat}, n_3\epsilon\mathsf{Nat}, v_2\epsilon\mathsf{vec}(U, n_2)), v_3\epsilon\mathsf{vec}(U, n_3)$. We want to show

$\forall v_1.(v_1\epsilon\mathsf{vec}(U, n_1) \to \mathsf{app}\ n_1\ (n_2+n_3)\ v_1\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (n_1+n_2)\ n_3\ (\mathsf{app}\ n_1\ n_2\ v_1\ v_2)\ v_3)$.

Let $P := \iota(y;z).(\mathsf{app}\ z\ (n_2+n_3)\ y\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (z+n_2)\ n_3\ (\mathsf{app}\ z\ n_2\ y\ v_2)\ v_3)$. We instantiate the C in $\mathsf{Ind}(U, n_1)$ with P, by comprehension we have

$(\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (y;m)\epsilon P \to (\mathsf{cons}\ m\ u\ y; \mathsf{S}m)\epsilon P)) \to (\mathsf{nil};0)\epsilon P \to \forall l.(l\epsilon\mathsf{vec}(U, n) \to (l;n)\epsilon P)$.

So we just need to prove base case:

$$\mathsf{app}\ 0\ (n_2+n_3)\ \mathsf{nil}\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (0+n_2)\ n_3\ (\mathsf{app}\ 0\ n_2\ \mathsf{nil}\ v_2)\ v_3$$

and step case:

$\forall y.\forall m.\forall u.(m\epsilon\mathsf{Nat} \to u\epsilon U \to (\mathsf{app}\ m\ (n_2+n_3)\ y\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (m+n_2)\ n_3\ (\mathsf{app}\ m\ n_2\ y\ v_2)\ v_3) \to (\mathsf{app}\ \mathsf{S}m\ (n_2+n_3)\ (\mathsf{cons}\ m\ u\ y)\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (\mathsf{S}m+n_2)\ n_3\ (\mathsf{app}\ \mathsf{S}m\ n_2\ (\mathsf{cons}\ m\ u\ y)\ v_2)\ v_3))$.

For the base case, $\mathsf{app}\ 0\ (n_2+n_3)\ \mathsf{nil}\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) \to_\beta^* \mathsf{app}\ n_2\ n_3\ v_2\ v_3 \leftarrow_\beta^* \mathsf{app}\ (0+n_2)\ n_3\ (\mathsf{app}\ 0\ n_2\ \mathsf{nil}\ v_2)\ v_3$. For the step case, we assume $\mathsf{app}\ m\ (n_2+n_3)\ y\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (m+n_2)\ n_3\ (\mathsf{app}\ m\ n_2\ y\ v_2)\ v_3$(IH), we want to show

$\mathsf{app}\ \mathsf{S}m\ (n_2 + n_3)\ (\mathsf{cons}\ m\ u\ y)\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) =$

$\mathsf{app}\ (\mathsf{S}m + n_2)\ n_3\ (\mathsf{app}\ \mathsf{S}m\ n_2\ (\mathsf{cons}\ m\ u\ y)\ v_2)\ v_3$(Goal).

We know that $\mathsf{app}\ \mathsf{S}m\ (n_2 + n_3)\ (\mathsf{cons}\ m\ u\ y)\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) \to_\beta^* \mathsf{cons}(m + n_2 + n_3)\ u\ (\mathsf{app}\ m\ (n_2 + n_3)\ y\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3))$. The right hand side of the (Goal) can be reduced to $\mathsf{cons}(m + n_2 + n_3)\ u\ (\mathsf{app}\ (m + n_2)\ n_3\ (\mathsf{app}\ m\ n_2\ y\ v_2)\ v_3)$. So (IH) is enough to give us the (goal). □

6.6 Termination Analysis in System $\mathfrak{G}$

In this section, we will show that elements in the inductive defined set are solvable. A direct consequence of this result is that these elements is terminating with respect to head reduction.

6.6.1 Preliminary

The definitions, lemmas and theorems in this subsection are came from Barendregt's [5], Chapter 8.3.

Definition 96 (Solvability)**.**

- *A closed lambda term t, i.e.* $\mathrm{FV}(t) = \emptyset$*, is solvable if there exists $t_1, ..., t_n$ such that $tt_1...t_n =_\beta \lambda x.x$.*
- *An arbitrary term t is solvable if the closure $\lambda x_1...\lambda x_n.t$, where $\{x_1, ..., x_n\} =$* $\mathrm{FV}(t)$*, is solvable.*
- *t is unsolvable iff t is not solvable.*

Lemma 45. *Every term t is of the following forms:*

- *$\lambda x_1....\lambda x_n.xt_1...t_m$, where $n, m \geq 0$. It is called head normal form.*
- *$\lambda x_1....\lambda x_n.((\lambda y.t)t_1)...t_m$, where $m \geq 1, n \geq 0$ and $(\lambda y.t)t_1$ is called head redex.*

Definition 97 (Head Reduction)**.** *$t \to_h t'$ if t' is resulting from contracting the head redex of t.*

Theorem 37. *A term t has a head normal form iff it is terminating with respect to head reduction.*

Theorem 38 (Wadsworth)**.** *t is solvable iff t has a head normal form. In particular, all terms in normal forms are solvable, and unsolvable terms have no normal form.*

Theorem 39 (Genericity)**.** *For a unsolvable term t, if $t_1 t =_\beta t_2$, where t_2 in normal form, then for any t', we have $t_1 t' =_\beta t_2$.*

Unsolvable in general is computational irrelevance, thus it is reasonable to equate all unsolvable terms.

Definition 98 (Omega-Reduction)**.** *Let Ω be $(\lambda x.xx)\lambda x.xx$, then $t \to_\omega \Omega$ iff t is unsolvable and $t \not\equiv \Omega$.*

Theorem 40. *$\to_\beta \cup \to_\omega$ is Church-Rosser.*

6.6.2 Head Normalization

We add Omega-reduction as part of the term reduction in $\mathfrak{G}$. We now define another notion of contradictory: $\bot' := \forall x.x = \Omega$. Note that this will imply $\forall x.\forall y.x = y$, thus we can safely take it as contradictory.

Theorem 41. $\vdash \forall n.(n \epsilon \mathsf{Nat} \to (n = \Omega \to \bot'))$.

Proof. We will prove this by induction. Recall the induction principle:

$$\Pi C^1.(\forall y.((y \epsilon C) \to (\mathsf{S}y)\epsilon C)) \to 0\epsilon C \to \forall m.(m\epsilon\mathsf{Nat} \to m\epsilon C).$$

We instantiate C with $\iota z.(z = \Omega \to \bot')$, by comprehension, we then have $(\forall y.((y = \Omega \to \bot') \to (\mathsf{S}y = \Omega \to \bot')) \to (0 = \Omega \to \bot') \to \forall m.(m\epsilon\mathsf{Nat} \to (m = \Omega \to \bot'))$. It is enough to show that $0 = \Omega \to \bot'$ and $\mathsf{S}y = \Omega \to \bot'$. We know that for Scott numerals we have $0 := \lambda s.\lambda z.z$ and $\mathsf{S}y := \lambda s.\lambda z.sy$. Assume $0 = \Omega = \lambda x_1.\lambda x_2.\Omega$, let $F := \lambda u.u\ p\ q$. Assume $q\epsilon X^1$, then $F\ 0\epsilon X^1$ (since $F0 =_\beta q$). So $F\ (\lambda x_1.\lambda x_2.\Omega)\epsilon X^1$, thus $\Omega\epsilon X^1$. Thus we just show $\forall X^1.(q\epsilon X^1 \to \Omega\epsilon X^1)$, which means $\forall q.q = \Omega$. So $0 = \Omega \to \bot'$. Now let us show $\mathsf{S}y = \Omega \to \bot'$. Assume $\lambda s.\lambda z.sy = \Omega = \lambda x_1.\lambda x_2.\Omega$. Let $F := \lambda n.n\ (\lambda p.q)\ z$. Assume $q\epsilon X^1$, then $F\ (\lambda s.\lambda z.sy)\epsilon X^1$, thus $F\ (\lambda x_1.\lambda x_2.\Omega)\epsilon X^1$, meaning $\Omega\epsilon X^1$. So we just show $\Pi X^1.(q\epsilon X \to \Omega\epsilon X)$. Thus $\forall q.q = \Omega$. So $\mathsf{S}y = \Omega \to \bot'$. □

Above theorem implies that all the member of Nat has a head normal form and it can be generalized to show that the elements of inductive describable set are solvable. To see this, we prove the following meta-theorem.

Theorem 42 (Meta)**.** *If* $\vdash t\epsilon\mathsf{Nat}$*, then* $t \neq_{\beta,\omega} \Omega$*.*

Proof. By theorem 41, we know that $\vdash t = \Omega \to \bot$. We know that by the *conv* rule, if $t =_{\beta,\omega} t'$, then $\vdash t = t'$ in $\mathfrak{G}$. By contraposition, we have if $\not\vdash t = t'$, then $t \neq_{\beta,\omega} t'$. Since $\mathfrak{G}$ is consistent (theorem 28), we know that $\not\vdash t = \Omega$. So $t \neq_{\beta,\omega} \Omega$. □

6.6.3 Leibniz Equality in $\mathfrak{G}$

We know that by the *conv* rule, if $t =_{\beta,\eta,\omega} t'$, then $\vdash t = t'$ in $\mathfrak{G}$. It is natural to consider wether the other direction is the case, namely, to prove: if $\vdash t = t'$, then $t =_{\beta,\eta,\omega} t'$. By the contraposition, we need to prove: if $t \neq_{\beta,\eta,\omega} t'$, then $\not\vdash t = t'$. We conjecture that it is hard to prove this. Due to the genericity (theorem 39) property in lambda calculus. Oraclely, if t is solvable and t' is unsolvable, we can not define a lambda term F such that $Ft =_\beta x$ and $Ft' =_\beta y$. Because by genericity, we would have $Ft =_\beta y$, thus $x =_\beta y$, which is impossible for beta-reduction. However, when t, t' both are solvable and $t \neq_{\beta,\eta} t'$, then by the results of Coppo et al. [13], we can indeed define a lambda term F such that $Ft =_\beta x$ and $Ft' =_\beta y$. So we can derive $\vdash t = t' \to \bot$ in System $\mathfrak{G}$.

Theorem 43. *Assume t_1, t_2 are solvable terms. If $\vdash t_1 = t_2$ in $\mathfrak{G}$, then $t_1 =_{\beta\eta} t_2$.*

Proof. By contraposition, we want to prove: if $t_1 \neq_{\beta\eta} t_2$, then $\not\vdash t_1 = t_2$. Since t_1 and t_2 are *distinct*, then t_1 and t_2 are *separable*[13]. i.e. there exists a lambda term F such that $Ft_1 =_\beta x$ and $Ft_2 =_\beta y$. Thus we can derive $\vdash t = t' \to \bot$. Since $\mathfrak{G}$ is consistent, we have $\not\vdash t_1 = t_2$. $\square$

The developments in this section together with section 6.6.2 shows that if $\vdash t = t' \to \bot$ in $\mathfrak{G}$, then $t \neq_{\beta,\eta,\omega} t'$. And if t_1, t_2 are solvable terms, then $\vdash t_1 = t_2$ in $\mathfrak{G}$ implies $t_1 =_{\beta\eta} t_2$.

[13]See Barendregt's [5], Page 256

6.7 Summary

We present System $\mathfrak{G}$ and develop Peano's axioms and Vector encoding in System $\mathfrak{G}$ as evidents for its potentials. The usefulness of $\mathfrak{G}[t]$ is not obvious in this Chapter. In implementation, we make use of the reciprocity to derive inductive set based on algebraic data type definitions. The existence of $\mathfrak{G}[t]$ provides a way to understand polymorphic-dependent type through $\mathfrak{G}$. One difference between System $\mathfrak{G}$ and PTS style system is that computation at formula level is currently not possible in $\mathfrak{G}$, more research will be needed to explore this issue.

Compare to usual typed functional programming language, the set in System $\mathfrak{G}$ is more precise than the notion of type in typed functional programming language. Not surprisingly, it is impossible to fully automate the reasoning with $\mathfrak{G}$. However, a degree of automation is still possible, together with human guidence, it would be an attracting tool to have besides the usual type system. In fact, the implementation in next Chapter shows that it is possible to obtain such an system.

CHAPTER 7

IMPLEMENTATION AND FUTURE IMPROVEMENTS

We first define the logic implemented in Gottlob, which is an extension of $\mathfrak{G}$. Then we discuss the current implemented features of Gottlob. Finally, we discuss some possible improvements over the current implementation.

7.1 The Gottlob System

The logic in Gottlob system is an extension of System $\mathfrak{G}$ with Church's simple types [11] and maintain the comprehension scheme à la Takeuti.

Definition 99 (Syntax)**.**

Simple Types $\tau \ ::= \ \iota \mid o \mid \tau \to \tau'$

Lambda Terms $t \ ::= \ x \mid \lambda x.t \mid tt'$

PreFormula $F \ ::= x \mid \iota x.F \mid t\epsilon F \mid F \to F' \mid \forall x.F \mid FF' \mid Ft$

Proof $p \ ::= \ x \mid \mathrm{mp}\ p\ p' \mid \mathrm{inst}\ p\ t \mid \mathrm{cmp}\ p \mid \mathrm{ug}\ x.p \mid \mathrm{discharge}\ x : F.p$

Type Context $\Delta \ ::= \ \cdot \mid \Delta, x : \tau$

Proof Context $\Gamma \ ::= \ \cdot \mid \Gamma, x : F$

The intended meaning of type ι is individuals and type o is formula. With Church's simple type device, the set mentioned in previous chapter will be a preformula of type $\iota \to o$.

Definition 100 (Type Inference for PreFormula)**.**

$$\frac{}{\Delta \vdash t : \iota} \qquad \frac{x : \tau \in \Delta}{\Delta \vdash x : \tau} \qquad \frac{\Delta, x : \tau' \vdash F : \tau}{\Delta \vdash \iota x.F : \tau' \to \tau}$$

$$\frac{\Delta \vdash F : \iota \to \tau}{\Delta \vdash Ft : \tau} \qquad \frac{\Delta \vdash F_1 : o \quad \Delta \vdash F_2 : o}{\Delta \vdash F_1 \to F_2 : o} \qquad \frac{\Delta \vdash F : \iota \to o}{\Delta \vdash t\epsilon F : o}$$

$$\frac{\Delta \vdash F : \tau \to \tau' \quad \Delta \vdash F' : \tau}{\Delta \vdash FF' : \tau'} \qquad \frac{\Delta, x : \tau \vdash F : o}{\Delta \vdash \forall x.F : o}$$

Note that type inference for preformula is decidable. We call a preformula of type o well-formed formula.

Definition 101 (Proof Checking Rules). $\boxed{\Gamma \vdash p : F}$

$$\frac{\Gamma \vdash p : F \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \mathrm{ug}\ x.p : \forall x.F} \qquad \frac{\Gamma \vdash p : F_1 \quad F_1 \cong F_2}{\Gamma \vdash \mathrm{cmp}\ p : F_2}$$

$$\frac{(x : F) \in \Gamma}{\Gamma \vdash x : F} \qquad \frac{\Gamma \vdash p : \forall x.F \quad Q ::= t \mid F}{\Gamma \vdash \mathrm{inst}\ p\ Q : [Q/x]F}$$

$$\frac{\Gamma, a : F_1 \vdash p : F_2}{\Gamma \vdash \mathrm{discharge}\ a : F_1.p : F_1 \to F_2} \qquad \frac{\Gamma \vdash p : F_1 \to F_2 \quad \Gamma \vdash p' : F_1}{\Gamma \vdash \mathrm{mp}\ p\ p' : F_2}$$

We can see that the proof checking is actually simpler than the one in Section 6.2.1, Chapter 6. The proof checking rule is specifically designed so that given Γ and p, we can deduce F with $\Gamma \vdash p : F$.

Definition 102. *$F \cong F'$ iff one of the following holds.*

- *$F \equiv [t/x]F_1$ and $F' \equiv [t'/x]F_1$ with $t =_\beta t'$.*
- *$F \equiv t\epsilon(\iota x.F_1)$ and $F' \equiv [t/x]F_1$.*
- *$F \equiv \mathcal{C}[(\iota x.F_1)Q]$ and $F' \equiv \mathcal{C}[[Q/x]F_1]$, where $Q ::= t \mid F$ for some preformula context $\mathcal{C}$.*

We have seen the full specification of the logic in Gottlob. It is consider more flexible in the sense that now Leibniz equality can be defined as

$\text{Eq} : \iota \to \iota \to o := \iota a.\iota b.\forall C.a\epsilon C \to b\epsilon C$

Vector can be defined as

$\text{Vec} : (\iota \to o) \to \iota \to \iota \to o := \iota U.\iota a.\iota x.\forall V....$

The point is that with help of comprehension and simple types, we do not need to appeal to meta-level conventions like we did before in last Chapter. Gottlob is considered more expressive than System $\mathfrak{G}$ in the sense that now we can express set of set, namely, entity of type $(\iota \to o) \to o$, up to arbitrary hierarchy. We can still define an erasure from Gottlob to Girard's System **F** (erasing everything except entity of type o), thus not every formula is provable in Gottlob.

7.2 The Implemented Features of Gottlob

Gottlob is implemented in Haskell, a functional-imperative programming language. The total lines of Haskell code (loc) currently is about 2700. About 600 loc are from the parser and the pretty-printing module; about 400 loc are used to describe syntax tree; about 700 loc are used to implement the proof checker; the rest of the codes deal with program transformation and polymorphic type checking. The project is available through `https://github.com/Fermat/Gottlob`.

Logic. The logic in Gottlob is described in Section 7.1. We implement a simple version of constraints solving algorithm to check the well-formedness of a formula. Proofs are represented as objects but not functions. So in Gottlob, we do not use proof as program and we do not run proof as program. That is not to say we can not program with proof. As we will discuss later, there are many common proof patterns we want to capture, and in Gottlob, we can use a notion of *tactic* to capture the proof pattern. Basically, a tactic is a meta-level function that take in an object (lambda terms, formula or proof) and return a proof. In Gottlob, formula/set can not be defined by recursion/induction. So inductive formula or inductive predicate is not supported. The proof language is in natural deduction style, while still allow user to write a big proof term to prove a theroem if she/he prefer. The proof language is carefully designed such that Gottlob can infer the formula of a proof term.

Proof Pattern and Tactic. After the second iteration of the implementation, the author realized that treating proof as object although can simplify the proof checking process, the author has to write long proof most of the time even to prove simple lemmas like congruence of equality. Long proof greatly affects the readability, readability is one of the goals of designing Gottlob. We notice that this issue can be fixed by introducing user-defined tactics in the proofs. By tactic we mean a meta-program that can take in proofs/formulas/programs as arguments and produce a checkable proof. The idea is that there are many proof patterns that can not be easily captured by lemma, but can be captured reasonably by tactic. For example, we know that we can always construct a proof of $t_1 =_{\text{Leibniz}} t_2$ if t_1 can be evaluated to t_2. However, the notion of "t_1 can be evaluated to t_2" can not captured by the language, so everytime one want to prove $t_1 =_{\text{Leibniz}} t_2$, one would need to manually construct a proof of $t_1 =_{\text{Leibniz}} t_2$. It is easy to see that all these proofs are the same except we only vary t_1, t_2. This proof pattern can be captured by introducing a meta-program that takes in t_1, t_2 as argument and produce a of proof of $t_1 =_{\text{Leibniz}} t_2$. This meta-program does not need to be typed, because the correctness of its outputs will always be checked by the proof checker.

Since the logic in Gottlob is higher order logic à la Takeuti, it can capture some common proof patterns. For example, we know that for *any* formula $F(x)$ with x free in F, we know that we can always construct a proof of $\forall x.F(x) \to F(x)$. This pattern can be captured by the proof of $\forall C.\forall x.x \in C \to x \in C$. To obtain a proof of $\forall x.F_1(x) \to F_1(x)$, one just need to instantiate C with $\iota x.F_1(x)$, then by comprehension we can get a proof of $\forall x.F_1(x) \to F_1(x)$. So we think that higher order logic in combination with tactic provide a good way to capture proof patterns.

About Gottlob Program. We mentioned that the logic includes the untyped lambda calculus as its domain of individuals. And untyped lambda calculus is basic of the program in Gottlob. It is natural to concerns about the type descipline for the program. Empirically, we realize that type checking can capture an range of bugs without requiring the programmer to annotate the program. So we implement a version of Hindley-Milner polymorphic type inference based on [29]. Our type inference system can handle mutual recursive defined programs *naturally* similar to the style of Haskell. We want to emphasis that even polymorphic type inference is convenient, it does not capture all the bugs and certainly does not verify a program. One would need to use the logic of Gottlob to prove theorems about programs. Gottlob's logic system treat program as untype lambda calculus. We think it is appropriate, because one usually need to reason about the execution behavior of the program, not the type behavior of a program, so the type information for a program is not as relevant as one may think. So when the author write programs and prove properties about the programs in Gottlob, internally, it first type check the programs, and then elaborate the programs to untyped lambda calculus, which is the execution model of the program, then finally, the reasoning is performed on the elaborated lambda terms[1].

Pattern Matching and Algebraic Data Type. Pattern matching and algebraic data type are central in Gottlob. Program can be defined as a set of "equations" just like Haskell function definition. And within each equation, one can use the *case* expression to further pattern match on data. So it seems like Gottlob does support pattern matching. Internally, polymorphic type checking is first performed on functions defined by pattern matching. After type checking, Gottlob translate a set of definitions in to a single equivalent function defined by case-expression (this process is described in [42]), then further translate this function to lambda term. The translation from case-expression to lambda term is done with respect to Scott encoding scheme. So case expression is not primitive in the execution model.

The translation process of pattern matching only make sense when the data is Scott encoded data. So for each algebraic data type declaration, Gottlob construct the corresponding Scott-encoded lambda term for each data type constructor. Data type declaration is also used for automatically deriving the corresponding inductive defined set and the corresponding induction principle. Data type declaration is also

[1]So it feels like reasoning directly on the compiled programs.

used for the polymorphic type checking. Let us see a concrete example, the following code are the data declaration for list and the append function in Gottlob.

```
data List U where
    nil :: List U
    cons :: U -> List U -> List U
  deriving Ind
append nil l = l
append (cons u l') l = cons u (append l' l)
```

From the type annotation in the data type declaration, Gottlob will infer the type of append is $\forall U.\text{List } U \to \text{List } U \to \text{List } U$. And it will also infer that

nil $:= \lambda n.\lambda c.n$

cons $:= \lambda a_2.\lambda a_1.\lambda n.\lambda c.c\ a_2\ a_1$

List $: (\iota \to o) \to \iota \to o\ = \iota U.\iota x.\forall L...$

indList $:=\ p : \forall U.\forall L.\text{nil}\epsilon L\ U \to (\forall x.x\epsilon U \to \forall x0.x0\epsilon L\ U \to \mathsf{cons}\ x\ x0\epsilon LU) \to \forall x.x\epsilon\text{List } U \to x\epsilon\ L\ U$.

Note that the p in indList is the proof of induction principle, Gottlob will check that the proof p. Also, Gottlob will perform this process for *any* algebraic data type.

7.3 Future Improvements

There are a lot of rooms for improvements for Gottlob.

Equality Reasoning. We want to implement an automatic equality reasoning feature to relieve the burden of simple equality proofs. Gottlob uses Leibniz equality extensively, so it is quite cumbersome to construct simple proofs about equality even with the help of tactic. We would need to implement this feature at meta-level and generate checkable proof of equality, then we do not need to trust the equality reasoning engine. We think this feature will greatly simplify the current equality proof while still give the author enough information to know the underlying mechanism.

Reasoning about States. Some algorithms (for example, graph algorithms) are natural to describe with the help of state, to really demonstrate the usefulness of Gottlob, we would need to do a case study on verifying this kind of algorithm. So we would need to provide a form of monadic framework in Gottlob.

Polymorphic Type Checking. Currently, the type checking system for Gottlob can only handle rank-1 polymorphism. It would be interesting to explore possible extensions of the type checking system to support richer notion of types while not requiring extensive annotations.

On a more practical side, the author would need to think about the issues of compilation, I/O and efficiency issues. It would also be nice to have an interpretor-like environment for the author to interact with the Gottlob system.

BIBLIOGRAPHY

[1] M. Abadi and L. Cardelli. A Theory of Primitive Objects - Second-Order Systems. In *European Symposium on Programming (ESOP)*, pages 1–25, 1994.

[2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In G. Morrisett and T. Uustalu, editors, *International Conference on Functional Programming (ICFP)*, pages 185–196, 2013.

[3] K.Y. Ahn, T. Sheard, M. Fiore, and A.M. Pitts. System Fi. In *Typed Lambda Calculi and Applications*, pages 15–30. 2013.

[4] Z. Ariola and J. Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154 – 233, 1997.

[5] H. Barendregt. *The lambda calculus: Its syntax and semantics*, volume 103. North Holland, 1985.

[6] H. Barendregt. Lambda calculi with types, handbook of logic in computer science (vol. 2): background: computational structures, 1993.

[7] B. Barras. Sets in coq, coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010.

[8] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[9] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[10] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 33–46. ACM, 2014.

[11] A. Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[12] A. Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies).* Princeton University Press, Princeton, NJ, USA, 1985.

[13] M. Coppo, M. Dezani-Ciancaglini, and S. R. D. Rocca. (semi)-separability of finite sets of terms in scott's D_∞-models of the lambda-calculus. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*, pages 142–164, London, UK, UK, 1978. Springer-Verlag.

[14] T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, September 1989.

[15] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.

[16] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II.* North-Holland, 1972.

[17] G. Frege. The basic laws of arithmetic: Exposition of the system, translated and edited with an introduction by montgomery furth, 1967.

[18] P. Fu and A. Stump. Self Types for Dependently Typed Lambda Encodings, 2014. Extended version available from `http://homepage.cs.uiowa.edu/~pfu/document/papers/rta-tlca.pdf`.

[19] P. Fu, A. Stump, and J. Vaughan. A framework for internalizing relations into type theory. In *PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, 2011.

[20] H. Geuvers. Inductive and Coinductive Types with Iteration and Recursion. In B. Nordstrom, K. Petersson, and G. Plotkin, editors, *Informal proceedings of the 1992 workshop on Types for Proofs and Programs*, pages 183–207, 1994.

[21] H. Geuvers. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications (TLCA)*, pages 166–181, 2001.

[22] E. Gimenez. *Un calcul de constructions infinies et son application a la verification de systemes communicants.* PhD thesis, 1996.

[23] J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, 1972.

[24] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types.* Cambridge University Press, New York, NY, USA, 1989.

[25] T. Hardin. Confluence results for the pure strong categorical logic ccl. λ-calculi as subsystems of ccl. *Theoretical Computer Science*, 65(3):291–342, July 1989.

[26] W. S. Hatcher. *The logical foundations of mathematics*, volume 10. Pergamon Press Oxford, 1982.

[27] J. Hickey. Formal objects in type theory using very dependent types. In K. Bruce, editor, *In Foundations of Object Oriented Languages (FOOL) 3*, 1996.

[28] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, pages 29–60, 1969.

[29] M. P. Jones. Typing haskell in haskell. In *Haskell workshop*, volume 43, page 45, 1999.

[30] G. Kimmell, A. Stump, H. D. Eades III, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, and K. Y. Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 15–26. ACM, 2012.

[31] J.-L. Krivine. Lambda-calculus types and models. 2002.

[32] D. Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 460–469. IEEE, 1983.

[33] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples,, Italy, 1984.

[34] R. Matthes. *Extensions of system F by iteration and primitive recursion on monotone inductive types.* Herbert Utz Verlag, 1999.

[35] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[36] A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique.* PhD thesis, PhD thesis, Université Paris 7, 2001.

[37] T. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2:345–364, 1994.

[38] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, USA, July 1990.

[39] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In L. Cardelli, editor, *17th European Conference on Object-Oriented Programming (ECOOP)*, pages 201–224, 2003.

[40] M. Parigot. Programming with Proofs: A Second Order Type Theory. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP)*, pages 145–159, 1988.

[41] G. Peano. *Arithmetices principia: nova methodo.* Fratres Bocca, 1889.

[42] S. L. Peyton Jones. *The implementation of functional programming languages (prentice-hall international series in computer science).* Prentice-Hall, Inc., 1987.

[43] C. RAFFALLI. *L' Arithmétiques Fonctionnelle du Second Ordre avec Points Fixes.* PhD thesis, L'université Paris VII, 1994.

[44] J. Rehof. Strong normalization for non-structural subtyping via saturated sets. *Inf. Process. Lett.*, 58:157–162, May 1996.

[45] D. Schepler. bijective function implies equal types is provably inconsistent with functional extensionality in coq. message to the Coq Club mailing list, December 12, 2013.

[46] V. Sjöberg and A. Stump. Equality, Quasi-Implicit Products, and Large Eliminations. In B. Venneri, editor, *Workshop on Intersection Types and Related Systems (ITRS)*, 2010.

[47] M. Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1):120–127, 1995.

[48] G. Takeuti. *Proof Theory.* Volume 81 of Studies in logic and the foundations of mathematics, ISSN 0049-237X. North-Holland Publishing Company, 1975.

[49] The Coq Development Team. The coq proof assistant reference manual. *Version 8.3. INRIA*, 2010.

[50] J. A. Vaughan. *Aura: Programming with Authorization and Audit.* PhD thesis, University of Pennsylvania, Philadelphia, 2009.

[51] B. Werner. A Normalization Proof for an Impredicative Type System with Large Elimination over Integers. In B. Nordström, K. Petersson, and G. Plotkin, editors, *International Workshop on Types for Proofs and Programs (TYPES)*, pages 341–357, 1992.

[52] B. Werner. *Une théorie des constructions inductives.* PhD thesis, Université Paris VII, 1994.