Lambda Encoding, Types and Confluence

Peng Fu
Computer Science, The University of Iowa

Last edited: May 19, 2013

Abstract

We review *lambda encoded data* in both untyped and typed forms. Several methods to prove *confluence* are surveyed. To address the problems of Church encoding arise in dependent type, we propose a novel type system called Selfstar, which not only enable us to type both Scott encoding and Church encoding data, but also allow us to derive corresponding induction principle and case analysis principle.

1 Introduction

1.1 Backgrounds

Modern computer stores instructions as well as its data in the memory [42], leave the distinctions between data and instructions conceptually. Most programming language distincts the program and other data on which the program operate, but in LISP and its dialects, programs and data are essentially indistinguishable. Programs as data give one ability to manipulate programs, this leads to the idea of metaprogramming . Data as program have been less well-known, perhaps the exploition of buffer overflow [16] provides an intuitive example, where the attacker provides a large trunk of data (including the malicious code) as input for a program such that the malicious code get executed. The idea of data as program can be expressed more naturally with lambda calculus, for example, the data 2 can be encoded as a lambda term to express the idea of doing something twice. For example $2\ (f, a) = f(f(a))$ means the function 2 takes two arguments f, a, where f is a function, returning the result of applying f twice to a; while 2 as data can be operated on, say $\mathsf{Plus}\ (2, 1) = 3$. Though this idea of data as program is familiared in functional programming language, it has not been widely adopted to handle algebraic data type. One of the purposes of this report is to explore the possibility of this approach.

Lambda calculus provide a syntactic way to model programming language in the sense that function application and function construction can be expressed explicitly. For example, given a function(or program), denoted by symbol f, the lambda expression corresponds to this function is $\lambda x.f\ x$, while calling this function on an argument denoted by symbol a can be expressed by the lambda expression $(\lambda x.f\ x)\ a$. The only rule of reducing a lambda expression is called beta reduction, for our example, $(\lambda x.f\ x)\ a \to_\beta f\ a$. (This way of understanding lambda calculus is inspired from Frege's [18].) It seems this kind of syntactic abstraction is quite limited, but surprisingly it is strong enough to capture many powerful computational concepts such as recursion, which we will introduce later in this report. Due to its computational power, lambda calculus give rise to the paradigm of functional programming.

Type is introduced to programming language with the purpose of reducing bugs. It is hard to give a direct answer of what the meaning of type is. But we should be able to say that it provide a way to express certain assumptions about the programs and data, so that compiler can check wether these assumptions are met when composing them together. For example, when a programmer write a function with type $\mathsf{int} \to \mathsf{int}$, he would expect the program to take integer as input, and no need to worry about dealing with situation that giving a string as an input.

In functional programming languages like Haskell, Ocaml, programmer can be implicit about these kind of *type* assumption, the compiler can automatically deduce the *type* according to the definition of the function, e.g. the type of function $f\ n = n + 1$ could be automatic infered as $\mathsf{int} \to \mathsf{int}$. With the assumptions get heavy, say one want to write a function that takes in a number n and return an indexed string array of size n(notationally, $\Pi n : \mathsf{Num}.\mathsf{Array}(\mathsf{String}, n)$), it is often hard for the compiler to deduce the type from the program, in this case, certain amount of annotation is needed. With this notion of type, one will reasonable want that type is invariant during the execution of program. For example, let $f\ n = n + 1$, $g\ n = n + 2$, one would want the return value of $f\ (g\ 3)$ to have type int. This requirement reflects the confirmation of the type system and the actual execution, is a kind of *soundness* property, we will formulate it later as type preservation property.

1.2 Motivations

It is well known that natural number can be encoded as lambda terms using Church encoding [10] or Scott encoding (reported in [15]). So operations such as *plus*, *multiplication* can be performed by beta-reduction (syntactic substitution) on lambda terms. Not only for natural number, other interesting inductive data structures such as trees, lists, etc. ([6], chapter 11 in [23]) can also be represented in a similar fashion. For discussons on the prospect of adopting Scott encoding in functional programming, we refer to Stump [37] and Jansen et al.[29].

Through Curry-Howard correspondent, type in typed lambda calculi corresponds to the formula in intuitionistic logic, and typed term corresponds to the proof of its formula(type) [28]. Due to this feature, typed lambda calculi, especially dependent typed lambda calculi [30] have been included in the core language for interactive theorem provers such as Agda, Coq([9], [40]); and for experimental functional programming languages such as Epigram 2, Guru ([31], [38]). Another part of the core is the add-on data type system, where various forms of data, including but not limted to inductive [33], coinductive [21], non-positive [35] datatype, are taken as primitives. From language design's point of view, if one want to adopt a rigor design methodology and want to define a type safe functional core language, then it is necessary to show the core language definitions satisfies *type preservation* and *progress* [45]. This requires substantial amount of efforts to write proofs even though most of the proofs can be done by case analysis and induction. For minimal core language such as Barendregt's lambda cube [7], the type preservation argument is given. For core language defintion which extends the lambda cube(or extends part of it), it is necessary and practical to have a notion of datatype, but when datatype and pattern matching facillities are added to the core language, together with binders, bound variables, alpha-conversion problems [3], it will substantially complicate the type preservation argument. For an example of this, see the Standard ML definition [32] and the type preservation report of Standard ML by VanInwegen [41]. Furthermore, if one want the core language to be a total type theory, i.e. can be used for reasoning, then a *termination* argument is required in order to show *logical consistency* [19]. In this case, present of datatype make it hard to analyze the termination behavior of a *well-typed* term. In fact, the core language of theorem prover Coq, *Calculus of Inductive Construction*(CIC), which extends *Calculus of Construction* [12] with inductive datatype [13] and restricted recursion, the strong normalization for well-typed term is still a conjecture [24].

Above discussions leads to the consideration of lambda encoding data as an alternative to handle datatype. Church encoding data can be typed in system **F** [22], part of Barendregt's lambda cube, type preservation argument and strong normalization will not be an issue. The drawbacks of this approach, as summerized in [43], are ineffecient to define certain operation on dataype, e.g. the *predecessor*, *minus* function; induction principle is not derivable and unable to prove $0 \neq 1$. This gives reason to fallback to Gödel's system **T** (chapter 7 in [23]), which takes boolean and natural number as primitives. Scott encoding does not suffer the ineffeciency problem arised in Church encoding, so as functional programming langauge, Scott encoding seems to be a better fit than Church encoding [29]. Scott encoding was claimed to be typable in System **F** [1], but it is unclear how to type recursve functions on such encoding in System **F**.

We propose a novel type system called $\mathsf{Selfstar}$, which not only enable us to type Scott encoding and Church encoding data, but also allow us to derive corresponding induction principle and case analysis principle. This makes it a possible candidate for as core functional language.

1.3 Overveiw

Definitions of abstract reduction system, lambda calculus, simple types are given in section 2. We present Scott and Church numerals in both untype and typed forms in section 3. Dependent type system and the related problem with Church encoding are discussed in detail in section 4. Seciton 5, several methods to show *confluence* are given. We give an outline of proving confluence for the term system of Selfstar (Section 5.2.1). Relation of confluence to type preservation is discussed in Section 5.3. We present system Selfstar (Section 6.1), which not only enable us to type Scott encoding and Church encoding data, but also allow us to derive corresponding induction principle and case analysis principle.

2 Preliminaries

2.1 Abstract Reduction System

We first introduce some basic concepts about *abstract reduction system*, sometimes it is also called *term rewriting system*, *labelled transition systems*.

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, $\to_i$ also 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* $\stackrel{*}{\to}_i$, *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 Calculus

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].

2.3 Simple Types

We use $A, B, C, X, Y, Z, ...$ to denote type variable, $T, S, U...$ to denote any type.

Definition 6.
Type $T\ ::=\ X \mid T_1 \to T_2$
Context $\Gamma\ ::=\ \cdot \mid \Gamma, x : T$

We call $T_1 \to T_2$ *arrow type*, *Typing* is a procedure to associate a term with a type. Typing is usually described by a set of rules, indicating how to associate a term t with a type T in given context Γ, denoted by $\Gamma \vdash t : T$. We present *simply typed lambda calculus* below.

Definition 7.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ \mathit{Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}\ \mathit{Abs} \qquad \frac{\Gamma \vdash t : T_1 \to T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t\ t' : T_2}\ \mathit{App}$$

Simply typed lambda calculus provides a basic framework for many sophisticated type systems. It is quite restrictive from both logical and programming point of view, since logically it corresponds to minimal intuitionistic propositional logic [27] and it only accepts a small set of strong normalizing terms. It has two properties, namely, type preservation and strongly normalization. For proofs of these two theorem we refer to [36].

Theorem 1 (Type Preservation)**.** *If* $\Gamma \vdash t : T$ *and* $t \to_\beta t'$, *then* $\Gamma \vdash t' : T$.

Theorem 2. *If* $\Gamma \vdash t : T$, *then* t *is strongly normalizing.*

3 Lambda Encoding with Types

3.1 Church Encoding

Definition 8 (Church Numeral)**.**
$0\ := \lambda s.\lambda z.z$
$\mathsf{S}\ := \lambda n.\lambda s.\lambda z.s(n\ s\ z)$

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

Informally, we can interpret lambda term as both data and function, so instead of thinking data 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.

One can define a notion of *iterator* $\mathsf{It}\ n\ f\ t\ := n\ f\ t$. So $\mathsf{It}\ 0\ f\ t =_\beta t$ and $\mathsf{It}\ (\mathsf{S}\ u)\ f\ t =_\beta f(\mathsf{It}\ u\ f\ t)$. So now we can use iterator to define $\mathsf{Plus}\ n\ m := \mathsf{It}\ n\ \mathsf{S}\ m$.

3.2 Scott Encoding

Definition 9 (Scott Numeral)**.**
$0\ := \lambda s.\lambda z.z$
$\mathsf{S}\ := \lambda n.\lambda s.\lambda z.s\ n$

We can see $1\ := \lambda s.\lambda z.(s\ 0)$, $2\ := \lambda s.\lambda z.(s\ 1)$. One can define a notion of *recursor*. But before defining that, we give one 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 $\mathsf{Fix}\ g \to_\beta (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x)) \to_\beta g((\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))) =_\beta g\ (\mathsf{Fix}\ g)$.

Since fix point operator is expressable with a lambda expression, the direct consequence is we can define recursor: $\mathsf{Rec}\ := \mathsf{Fix}\ \lambda r.\lambda n.\lambda f.\lambda v.n\ (\lambda m.f\ (r\ m\ f\ v)\ m)\ v$. So 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 [23].

3.3 Church Encoding in System F

System **F** is an extension of simply typed lambda calculus, the only addition is a *polymorphic type* $\forall X.T$. Note that here $\forall$ is a binder. The additional typing rules are follows:

$$\frac{\Gamma \vdash t : T \quad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X.T}\ Gen \qquad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t : [T'/X]T}\ Inst$$

$X \notin FV(\Gamma)$ means X is not a free type variable in the types of the typing context Γ. For example, given above typing rule, we can asscociate identity function with a polymorphic type, i.e. $\cdot \vdash \lambda x.x : \forall X.X \to X$. And we also have $\cdot \vdash \lambda x.x : T \to T$ for any type T.

We define $\mathsf{Nat}\ := \forall X.(X \to X) \to X \to X$. One can type the constructors 0 and S as following.

$$\cfrac{\cfrac{\cfrac{}{s : X \to X, z : X \vdash z : X}}{\cdot \vdash \lambda s.\lambda z.z : (X \to X) \to X \to X}}{\cdot \vdash \lambda s.\lambda z.z : \forall X.(X \to X) \to X \to X}$$

For space reason, we only list $\cdot \vdash 0 : \mathsf{Nat}$, similarly one will can type:
$\cdot \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}$
$\cdot \vdash \mathsf{It} : \forall X.\mathsf{Nat} \to (X \to X) \to X \to X$
$\cdot \vdash \mathsf{Plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$

System **F** and Church encoding fit together really well, indeed, being able to define inductive data type within the type system is one of the motivations for devising system **F**[23]. Through Curry-Howard correspondent, one can view types in system **F** as intuitionistic proposition, the quantification proposition $\forall X.T$ is considered *impredicative* in the sense that X can be instantiated by any proposition, including itself, terms in system **F** becomes proof for the proposition. System **F** is also type preserving and strongly normalizing [23].

3.4 Scott Encoding with Recursive Types

It is not obvious to type Scott encoding with system **F**. We extend simply typed lambda calculus with *recursive types* $\mu X.T$. Note that here μ is a binder. The additional typing rules are follows:

$$\frac{\Gamma \vdash t : [\mu X.T/X]T}{\Gamma \vdash t : \mu X.T}\ \mathit{Fold} \qquad \frac{\Gamma \vdash t : \mu X.T}{\Gamma \vdash t : [\mu X.T/X]T}\ \mathit{unFold}$$

With recursive type, define $\mathsf{Nat} := \mu X.(X \to U) \to U \to U$ for any type U. We introduce a notation $T \sim T'$ to mean there exist a derivation from $\Gamma \vdash t : T$ to $\Gamma \vdash t : T'$, is called *morphing* relation. Thus $\mathsf{Nat} \sim (\mathsf{Nat} \to U) \to U \to U$.

$$\cfrac{\cfrac{\overline{s : \mathsf{Nat} \to U, z : U \vdash z : U}}{\cdot \vdash \lambda s.\lambda z.z : (\mathsf{Nat} \to U) \to U \to U}}{\cdot \vdash \lambda s.\lambda z.z : \mathsf{Nat}} \qquad \cfrac{\cfrac{\overline{n : \mathsf{Nat}, s : \mathsf{Nat} \to U, z : U \vdash s\ n : U}}{n : \mathsf{Nat} \vdash \lambda s.\lambda z.s\ n : (\mathsf{Nat} \to U) \to U \to U}}{n : \mathsf{Nat} \vdash \lambda s.\lambda z.s\ n : \mathsf{Nat}}$$

$\cdot \vdash \mathsf{Fix} : (U \to U) \to U$ for any type U.
$\cdot \vdash \mathsf{Rec} : \mathsf{Nat} \to (U \to \mathsf{Nat} \to U) \to U \to U$.
$\cdot \vdash \mathsf{Plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.

Recursive types is powerful enough to capture the typing for Church encoding, define $\mathsf{Nat} := \mu X.(X \to X) \to X \to X$. Thus $\mathsf{Nat} \sim (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$.

$$\cfrac{\cfrac{\overline{s : \mathsf{Nat} \to \mathsf{Nat}, z : \mathsf{Nat} \vdash z : \mathsf{Nat}}}{\cdot \vdash \lambda s.\lambda z.z : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}}}{\cdot \vdash \lambda s.\lambda z.z : \mathsf{Nat}} \qquad \cfrac{\cfrac{\overline{n : \mathsf{Nat}, s : \mathsf{Nat} \to \mathsf{Nat}, z : \mathsf{Nat} \vdash s\ (n\ s\ z) : \mathsf{Nat}}}{n : \mathsf{Nat} \vdash \lambda s.\lambda z.s\ (n\ s\ z) : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}}}{n : \mathsf{Nat} \vdash \lambda s.\lambda z.s\ (n\ s\ z) : \mathsf{Nat}}$$

$\cdot \vdash \mathsf{It} : \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$. We can see that here, because of lack of support for polymorphic type, the iterator for Church encoding numerals can only deal with numerals. Note that recursive types can not be interpreted as formula under Curry-Howard correspondent.

Recursive types and its denotational semantics have been studied extensively in [44], [7]. The recursive type system is type preserving but not strongly normalizing.

4 Dependent Type

In order to enable type to mention terms, we extend the types of system **F** with *dependent type*(product type) $\Pi x : T.T'$ and *index type* $T\ t$. The additional typing rules are:

$$\frac{\Gamma, x : T' \vdash t : T \quad x \in FV(T)}{\Gamma \vdash \lambda x.t : \Pi x : T'.T}\ \mathit{Pi} \qquad \frac{\Gamma \vdash t : \Pi x : T'.T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ t' : [t'/x]T}\ \mathit{Elim}$$

$$\frac{\Gamma \vdash t : [t_1/x]T \quad t_1 =_\beta t_2}{\Gamma \vdash t : [t_2/x]T}\ \mathit{Conv}$$

So far the typing rule does not prevent us to write things like $(T_1 \to T_2)x$, so we introduce *kind*(denoted by κ) and the process of *kinding* to regulate the type we write. We extend the notion of context and change the form of the type $\forall X.T$ in system **F** to $\forall X : \kappa.T$ to allow a finer classification of type. We say type T has kind κ under the context Γ, denoted by $\Gamma \vdash T : \kappa$.

Definition 10 (Kind and Kinding). *Kind* $\kappa := * \mid \xi x : T.\kappa$
Context $\Gamma := \cdot \mid \Gamma, x : T \mid \Gamma, X : \kappa$

$$\frac{X : \kappa \in \Gamma}{\Gamma \vdash X : \kappa}\ \textit{K-Var} \qquad \frac{\Gamma \vdash S : \xi x : T'.\kappa \quad \Gamma \vdash t' : T' \quad \Gamma \vdash T' : *}{\Gamma \vdash S\ t' : [t'/x]\kappa}\ \textit{K-App}$$

$$\frac{\Gamma \vdash T' : * \quad \Gamma, x : T' \vdash T : *}{\Gamma \vdash \Pi x : T'.T : *}\ \textit{K-Pi} \qquad \frac{\Gamma, X : \kappa \vdash T : *}{\Gamma \vdash \forall X : \kappa.T : *}\ \textit{K-Forall}$$

$$\frac{\Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \to T_2 : *}\ \textit{K-Arrow}$$

Note that the ξ in $\xi x : T.\kappa$ is a binder, with its scope in κ.

Definition 11 (Revised System **F** Typing)**.**

$$\frac{\Gamma, X : \kappa \vdash t : T \quad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X : \kappa.T}\ \textit{Gen} \qquad \frac{\Gamma \vdash t : \forall X : \kappa.T \quad \Gamma \vdash T' : \kappa}{\Gamma \vdash t : [T'/X]T}\ \textit{Inst}$$

Now let us see an example. We already seen Church numeral Nat with operation Plus can be encoded in System **F**. With dependent type, we can do some lightweight external reasoning on our Church numerals. One can express *Leibniz's law* as: $\mathsf{Eq}\ [A]\ x\ y := \forall C : (\xi z : A.*).C\ x \to C\ y$. It reads as: given any x and y, they are in the relation $\mathsf{Eq}(x, y)$ if, given any predicate C, if $C(x)$, then $C(y)$ [17]. Now we can show $\cdot \vdash \lambda x.x : \mathsf{Eq}\ [\mathsf{Nat}]\ (\mathsf{Plus}\ 1\ 1)\ 2$. It is derivable because the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{C : (\xi z : \mathsf{Nat}.*), x : C\ (\mathsf{Plus}\ 1\ 1) \vdash x : C\ (\mathsf{Plus}\ 1\ 1)}\ \textit{Var} \qquad (\mathsf{Plus}\ 1\ 1) =_\beta 2}{C : (\xi z : \mathsf{Nat}.*), x : C\ (\mathsf{Plus}\ 1\ 1) \vdash x : C\ 2}\ \textit{Conv}}{C : (\xi z : \mathsf{Nat}.*) \vdash \lambda x.x : C\ (\mathsf{Plus}\ 1\ 1) \to C\ 2}\ \textit{Abs}}{\cdot \vdash \lambda x.x : \forall C : (\xi z : \mathsf{Nat}.*).C\ (\mathsf{Plus}\ 1\ 1) \to C\ 2}\ \textit{Gen}}{\cdot \vdash \lambda x.x : \mathsf{Eq}\ [\mathsf{Nat}]\ (\mathsf{Plus}\ 1\ 1)\ 2}\ \textit{Def}$$

Note that the last step is by definition of Eq. Similarly, the following is a derivation of $A : *, a : A \vdash \lambda x.x : \mathsf{Eq}\ [A]\ a\ a$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A : *, a : A, C : (\xi z : A.*), x : C\ a \vdash x : C\ a}\ \textit{Var}}{A : *, a : A, C : (\xi z : A.*) \vdash \lambda x.x : C\ a \to C\ a}\ \textit{Abs}}{A : *, a : A \vdash \lambda x.x : \forall C : (\xi z : A.*).C\ a \to C\ a}\ \textit{Gen}}{A : *, a : A \vdash \lambda x.x : \mathsf{Eq}\ [A]\ a\ a}\ \textit{Def}}{A : * \vdash \lambda a.\lambda x.x : \Pi a : A.\mathsf{Eq}\ [A]\ a\ a}\ \textit{Abs}}{\cdot \vdash \lambda a.\lambda x.x : \forall A : *.\Pi a : A.\mathsf{Eq}\ [A]\ a\ a}\ \textit{Gen}$$

The derivation above can be viewed as a *proof* of *reflexitive* for the Eq(Namely, the formula $\forall A : *.\Pi a : A.\mathsf{Eq}\ [A]\ a\ a$); or it can be viewed as a procedure to associate the type $\forall A : *.\Pi a : A.\mathsf{Eq}\ [A]\ a\ a$ to a lambda term $\lambda a.\lambda x.x$. Since under Curry-Howard correspondent, dependent type can be interpreted as intuitionistic formula [7]. One can also derive a proof for $\forall A : *.\Pi a : A.\Pi b : A.\forall B : (\xi x : A.*).B\ a \to (\mathsf{Eq}\ [A]\ a\ b) \to B\ b$, which is called the *substitution property* for Leibniz's equality.

We have seen we can prove some very basic properties about Church numerals with the operation Plus and the relation Eq. For functions like Plus, one would also like to prove properties about commutativity, associativity etc. For such properties, the proof normally will involve induction argument, but it is known that induction is not derivable in second order dependent type system [20]. Note that induction principle can be expressed as $\mathsf{Id} := \forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to \Pi x : \mathsf{Nat}.P\ x$. So this means there does not exist any derivation and term t such that $\cdot \vdash t : \mathsf{Id}$. Because, informally, we can only have

$x : \mathsf{Nat} \vdash 0 : \forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to P\ 0$
$x : \mathsf{Nat} \vdash \bar{n} : \forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to P\ \bar{n}$, for any Church numerals $\bar{n}$
Here we can see that the dependent type system can not capture the notion of *for any Church numerals* $\bar{n}$, since it involves a meta-level quantification. This problem is also discussed in Coquand's [11]. In later section, we propose Selfstar, with the help of recursive definition and self type mechanism, we can effectively capture this kind of meta-level quantification.

The second problem is that in dependent type with Church encoding [43], $0 \neq 1$ is not derivable. Note that $0 \neq 1$ can be represented as $(\mathsf{Eq}\ [\mathsf{Nat}]\ 0\ 1) \to \bot$, where $\bot := \forall Q : *.Q$. In intuitionistic logic, $\neg A$ is expressed as $A \to \bot$. $\bot$ means contradiction, can be encoded in system **F** as $\forall Q : *.Q$, because there is no derivation such that $\cdot \vdash t : \forall Q : *.Q$ [23]. Because the non-derivability of $\bot$, $(\mathsf{Eq}\ [\mathsf{Nat}]\ 0\ 1) \to \bot$ become underivable. Thus $0 \neq 1$ is not a theorem in dependent type system.

In a sense these problems are not surprising, that is why in *Peano arithmetic* and *Heyting arithmetic* [34], [26], both of these system have numbers as primitives, induction, $0 \neq 1$ and many others as axioms.

5 Confluence

We seen the use of *reduction*$(\to_\beta)$ to convey a notion of *equation*$(=_\beta)$. Informally, the equation $a = b$ means a and b denotes the same thing; while the reduction $a \to b$ convey how one can obtain the expression b from a, even though both a and b denote the same thing. So to interpret the equation $\mathsf{S}\ 0\ +\ (\mathsf{S}\ \mathsf{S}\ 0)\ =\ \mathsf{S}\ \mathsf{S}\ \mathsf{S}\ 0$ by saying the expressions at both side denote the same thing is uninteresting; but $\mathsf{S}\ 0\ +\ (\mathsf{S}\ \mathsf{S}\ 0) \to 0\ +\ (\mathsf{S}\ \mathsf{S}\ \mathsf{S}\ 0) \to \mathsf{S}\ \mathsf{S}\ \mathsf{S}\ 0$ shows how one can obtain the expression $\mathsf{S}\ \mathsf{S}\ \mathsf{S}\ 0$ from the expression $\mathsf{S}\ 0\ +\ (\mathsf{S}\ \mathsf{S}\ 0)$ by applying appropriate rules. The difference between reduction and equation is discussed in Girard's ([23], chapter 1), for a philosophical perspective on this distinction(*sense and denotation* distinction), we refer to Frege's [18].

Beta reduction $\to_\beta$ will be analyzed in this section. We first identify the *confluent* property and *Church-Rosser* property in abstract reduction system (similar treatment can be found at [8], [4]) and illustrate some consequences of confluence and Church-Rosser, then we survey three methods to show $\to_\beta$ reduction is confluent.

Definition 12. *Given an abstract reduction system* $(\mathcal{A}, \{\to_i\}_{i \in \mathcal{I}})$*, let* $\to$ *denote* $\bigcup_{i \in \mathcal{I}} \to_i$*, let* $=$ *denote the equivalence relation generated by* $\to$*.*

- *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:

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

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

Proof. Assume the same notation as defintion 12.

"$\Leftarrow$": 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:

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

□

The definition of = depends on $\twoheadrightarrow$, the definition of $\twoheadrightarrow$ depends on $\to$, 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 13. *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.*

5.1 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 14 (Diamond Property)**.** *Given an abstract reduction system $(\mathcal{A}, \{\to_i\}_{i \in \mathcal{I}})$, it has diamond property if:*

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

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

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.

5.1.1 Confluence of 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 15 (Parallel Reduction)**.**

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

Intuitively, parallel reduction allows us to contract many beta redex(or not contracting at all) in once 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 $t_2 \equiv [t_5''/x]t_4'' \Rightarrow_\beta [t_6/x]t_7 \equiv t_3$.

Note: Careful readers are recommended to draw the corresponding diagrams while reading this proof.

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 3. *$\to_\beta$ reduction is confluent.*

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

5.1.2 Takahashi's Method

Based on the notion of parallel reduction, Takahashi [39] observed, instead of trying to prove $\Rightarrow_\beta$ has diamond property, one can prove a stronger property.

Definition 16. *$\Rightarrow_\beta$ is said to satisfy triangle property if: $t \Rightarrow_\beta t'$ implies $t' \Rightarrow_\beta t^*$, where t^*(we called it parallel contraction) is defined as:*
$x^* \ := \ x$.
$(\lambda x.t)^* \ := \ \lambda x.t^*$.
$(t_1 \ t_2)^* \ := \ t_1^* \ t_2^*$ if $t_1 \ t_2$ is not a beta redex.
$((\lambda x.t_1) \ t_2)^* \ := \ [t_2^*/x]t_1^*$.

$$\begin{tikzcd}
t \arrow[dd, mapsto] \arrow[dr, Rightarrow] & \\
& t' \arrow[dl, Rightarrow, dashed] \\
t^* &
\end{tikzcd}$$

One can see that the definition of t^* only depends on t and $\_^*$ is really a recursively defined function that contract all the redex in t, so once we prove $\Rightarrow_\beta$ has triangle property(name from [8]), that will implies the diamond property.

Lemma 7. *If $\Rightarrow_\beta$ has triangle property, then it has diamond property.*

Proof.

$$\begin{tikzcd}
& t \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] \arrow[dd, mapsto] & \\
t_1 \arrow[dr, Rightarrow, dashed] & & t_2 \arrow[dl, Rightarrow, dashed] \\
& t^* &
\end{tikzcd}$$

□

Lemma 8. $\Rightarrow_\beta$ *has triangle property.*

Proof. Assume $t \Rightarrow_\beta t'$, we prove this by induction on the derivation of $t \Rightarrow_\beta t'$.

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

We need to show $t \Rightarrow_\beta t^*$. This can be proved by induction on the form of t, we will not go through the proof here.

Case: $\dfrac{t_1 \Rightarrow_\beta t_1'}{\lambda x.t_1 \Rightarrow_\beta \lambda x.t_1'}$

t is of the form $\lambda x.t_1$, where $t_1 \Rightarrow_\beta t_1'$; t' is of the form $\lambda x.t_1'$. By induction, there exist a reduction $t_1' \Rightarrow_\beta t_1^*$. Thus there is a reduction $\lambda x.t_1' \Rightarrow_\beta \lambda x.t_1^* \equiv (\lambda x.t_1)^*$.

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

t is of the form $(\lambda x.t_4)t_5$, t' is of the form $[t_5'/x]t_4'$, $t_4 \Rightarrow_\beta t_4'$ and $t_5 \Rightarrow_\beta t_5'$. By induction, there is a reduction $t_4' \Rightarrow_\beta t_4^*$ and $t_5' \Rightarrow_\beta t_5^*$. Thus there is a reduction $[t_5'/x]t_4' \Rightarrow_\beta [t_5^*/x]t_4^* \equiv ((\lambda x.t_4)\ t_5)^*$(lemma 4).

Case: $\dfrac{t_4 \Rightarrow_\beta t_4' \quad t_5 \Rightarrow_\beta t_5'}{t_4 t_5 \Rightarrow_\beta t_4' t_5'}$

t is of the form $t_4\ t_5$, t' is of the form $t_4'\ t_5'$, $t_4 \Rightarrow_\beta t_4'$ and $t_5 \Rightarrow_\beta t_5'$.

Assume $t_4 \equiv \lambda x.t_6$, then t_4' must be of the form $\lambda x.t_6'$ with $t_6 \Rightarrow_\beta t_6'$. By induction, there is a reduction $t_6' \Rightarrow_\beta t_6^*$ and $t_5' \Rightarrow_\beta t_5^*$. So $t_4'\ t_5' \equiv (\lambda x.t_6')\ t_5' \Rightarrow_\beta [t_5^*/x]t_6^* \equiv ((\lambda x.t_6)\ t_5)^* \equiv (t_4\ t_5)^*$.

Assume t_4 is not of the form $\lambda x.t_6$. By induction, there is a reduction $t_4' \Rightarrow_\beta t_4^*$ and $t_5' \Rightarrow_\beta t_5^*$. Thus there is a reduction $t_4' t_5' \Rightarrow_\beta t_4^*\ t_5^* \equiv (t_4\ t_5)^*$.

□

5.1.3 Barendregt's Labelling Method

Barendregt provide a method to prove the confluence of beta reduction for lambda calculus without appeal to diamond property [5], which has an advantage over Tait-Martin Löf's(and Takahashi's) method in the sense that one does not need to formulate the parallel reduction.

The new concepts involved are *labelled terms* and *labelled reduction*, both of which are extension of the usual terms and reduction.

Definition 17 (Labelled Terms)**.**
$t \ ::= \ x \mid \lambda x.t \mid tt' \mid (\underline{\lambda} x.t)t'$

We simply label certain beta redexes. Note that $\underline{\lambda} x.t$ is not a well-formed labelled term, but $(\underline{\lambda} x.t)t'$ is a well-formed labelled term. The labelled beta reduction extends the usual beta reduction *naturally* in the sense that it can reduce the labelled beta redex.

Definition 18 (Labelled Beta Reduction)**.**

$$\frac{}{(\lambda x.t)t' \to_{\underline{\beta}} [t'/x]t} \quad \frac{}{(\underline{\lambda} x.t)t' \to_{\underline{\beta}} [t'/x]t} \quad \frac{t \to_{\underline{\beta}} t'}{\lambda x.t \to_{\underline{\beta}} \lambda x.t'} \quad \frac{t \to_{\underline{\beta}} t''}{tt' \to_{\underline{\beta}} t''t'}$$

$$\frac{t' \to_{\underline{\beta}} t''}{tt' \to_{\underline{\beta}} tt''} \quad \frac{u \to_{\underline{\beta}} u'}{(\underline{\lambda} x.u)t' \to_{\underline{\beta}} (\underline{\lambda} x.u')t'} \quad \frac{t' \to_{\underline{\beta}} t''}{(\underline{\lambda} x.u)t' \to_{\underline{\beta}} (\underline{\lambda} x.u)t''}$$

It is natural to make sure that: if t is a well-formed labelled term and $t \to_{\underline{\beta}} t'$, then t' is also a well-formed labelled term. We can do this by induction on the derivation of $t \to_{\underline{\beta}} t'$. We will use $\underline{\Lambda}$ to denote the set of all labelled terms, Λ to denote the set of unlabelled terms. So $\Lambda \subset \underline{\Lambda}$. As usual, $\twoheadrightarrow_{\underline{\beta}}$ denotes the reflexive and transitive closure of $\to_{\underline{\beta}}$. $\to_{\underline{\beta}}$ is defined on terms in $\underline{\Lambda}$. Note that $\to_\beta \subseteq \to_{\underline{\beta}}$.

Definition 19 (Erasure)**.** *We define erasure function $e : \underline{\Lambda} \to \Lambda$ as below:*

$e(x) \ := x$
$e(tt') \ := e(t)e(t')$
$e(\lambda x.t') \ := \lambda x.e(t')$
$e((\underline{\lambda} x.t)t') \ := (\lambda x.e(t))e(t')$
graphically denoted by $\to_e$

Definition 20 (Contraction)**.** *We define a contraction function $\phi : \underline{\Lambda} \to \Lambda$ as below:*

$\phi(x) \ := x$
$\phi(tt') \ := \phi(t)\phi(t')$
$\phi(\lambda x.t') \ := \lambda x.\phi(t')$
$\phi((\underline{\lambda} x.t)t') \ := [\phi(t')/x]\phi(t)$
graphically denoted by $\to_\phi$

The erasure recursively remove all the labels in a term $t \in \underline{\Lambda}$ without changing the structure of the term. The contraction functions recursively reduce all the labelled redexes in a labelled term.

Lemma 9. *If $t_1 \to_\beta t_2$ and $t_1' \to_e t_1$, then there exist t_2' such that $t_1' \to_{\underline{\beta}} t_2'$, and $t_2' \to_e t_2$.*

$$\begin{tikzcd}
t_1' \arrow[r, dashed, "\underline{\beta}"'] \arrow[d, "e"] & t_2' \arrow[d, dashed, "e"] \\
t_1 \arrow[r, "\beta"'] & t_2
\end{tikzcd}$$

Proof. If $t_1 \to_\beta t_2$, then t_2 is obtained by contracting one redex Δ in t_1. We reduce Δ(either labelled or unlabelled) in t_1' we get t_2'(recalled that $\to_\beta \subseteq \to_{\underline{\beta}}$), which has $e(t_2') = t_2$. $\square$

Lemma 10.

$$\begin{tikzcd}
t_1' \arrow[r, dashed, two heads, "\underline{\beta}"'] \arrow[d, "e"] & t_2' \arrow[d, dashed, "e"] \\
t_1 \arrow[r, two heads, "\beta"'] & t_2
\end{tikzcd}$$

Proof. Using lemma 9. By transitivity. $\square$

Lemma 11. $\phi([t'/x]t) = [\phi(t')/x]\phi(t)$.

Proof. By induction on the structure of labelled term t. $\square$

Lemma 12.

$$\begin{tikzcd}
t_1 \arrow[r, "\underline{\beta}"'] \arrow[d, "\phi"] & t_2 \arrow[d, "\phi"] \\
\phi(t_1) \arrow[r, dashed, two heads, "\beta"'] & \phi(t_2)
\end{tikzcd}$$

Proof. By induction on the derivation of $t_1 \to_{\underline{\beta}} t_2$. Using lemma 11. $\square$

Lemma 13.

$$
\begin{tikzcd}
 & t \arrow[dl, "e"'] \arrow[dr, "\phi"] & \\
e(t) \arrow[rr, dashed, two heads, "\beta"'] & & \phi(t)
\end{tikzcd}
$$

Proof. By induction on the structure of t. □

Lemma 14 (Strip Lemma)**.**

$$
\begin{tikzcd}
 & t \arrow[dl, "\beta"'] \arrow[dr, two heads, "\beta"] & \\
t_1 \arrow[dr, dashed, two heads, "\beta"'] & & t_2 \arrow[dl, dashed, two heads, "\beta"] \\
 & t' &
\end{tikzcd}
$$

Proof. Let t_1 be the result of reducing the redex Δ in t. Let $t_3 \in \underline{\Lambda}$ be the term obtained from t by indexing Δ. So $\phi(t_3) \equiv t_1$. By the following diagram:

$$
\begin{tikzcd}
 & t \arrow[dl, "\beta"'] \arrow[dr, two heads, "\beta"] & \\
t_1 \arrow[dr, dashed, two heads, "\beta"'] & t_3 \arrow[u, "e"] \arrow[l, "\phi"] \arrow[dr, dashed, two heads, "\beta"] & t_2 \arrow[dl, dashed, two heads, "\underline{\beta}"'] \\
 & t' & t_4 \arrow[l, "\phi"] \arrow[u, dashed, "e"']
\end{tikzcd}
$$

□

Strip lemma implies confluence by simple diagram chasing. Thus we can conclude the confluence of lambda calculus. The above proof of strip lemma relies on the fact that: for any $t \to_\beta t_1$, there exist $t_3 \in \underline{\Lambda}$ such that $e(t_3) \equiv t$ and $\phi(t_3) \equiv t_1$. This limits the application of this method to the system that contains multiple kinds of redexes and reductions. For example lambda calculus extends with $\lambda x.t\ x \to_\eta t$. The term $\lambda x.(\lambda y.y\ z)\ x$ contains both beta redex and eta redex, contracting one will make the other disappear. So if the definition of ϕ and e unchanged, consider $\lambda x.(\lambda y.y\ z)\ x \to_\eta \lambda y.y\ z$. Since it does not constract a beta redex, if we let $t_3 \equiv \lambda x.(\lambda y.y\ z)\ x \in \underline{\Lambda}$, then $e(t_3) \equiv \lambda x.(\lambda y.y\ z)\ x$ and $\phi(t_3) \equiv \lambda x.(\lambda y.y\ z)\ x$. So $\phi(t_3) \not\equiv \lambda y.y\ z$ in this case. So this method can not directly generalize to deal with lambda calculus with beta and eta reductions, and also for reduction systems with multiple kinds of redexes and reductions.

5.2 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 15 (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* $\nu(a) \twoheadrightarrow_i \nu(b)$ $(\dagger)$

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 $(\dagger)$, $\nu(a) \twoheadrightarrow_i \nu(a')$ and $\nu(a) \twoheadrightarrow_i \nu(a'')$. Notice that $t \to_1^* t'$ implies $\nu(t) = \nu(t')$(By confluence and strong normalizing of $\to_1$). By confluence of $\to_i$, there exists b such that $\nu(a') \twoheadrightarrow_i b$ and $\nu(a'') \twoheadrightarrow_i b$. Since $\to_i, \to_1 \subseteq \twoheadrightarrow$, we got $a' \twoheadrightarrow \nu(a') \twoheadrightarrow b$ and $a'' \twoheadrightarrow \nu(a'') \twoheadrightarrow b$. Hence $\to$ is confluent.

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

□

Hardin's method reduce the confluence problem of $\to_1 \cup \to_2$ to $\to_i$, given the confluence and strong normalizing of $\to_1$, this make it possible to apply Tait-Martin-Löf's (Takahashi's) method to prove confluence of $\to_i$.

5.2.1 Local $\lambda\mu$ Calculus

We now show an applicaiton of Hardin's method on a concrete example, this example arise naturally in proving type preservation for Selfstar. The approach we adopt is similar to the one in [14]. The proofs are in the appendix.

Definition 21 (Local Lambda Mu Terms)**.**
Terms $t \ ::= \ x \mid \lambda x.t \mid tt' \mid \mu t$
Closure $\mu \ ::= \{x_i \mapsto t_i\}_{i \in \mathcal{I}}$

The closure is basically a set of recursively defined definitions. Let $\mathcal{I}$ be a finite nonempty index set. For $\{x_i \mapsto t_i\}_{i\in\mathcal{I}}$, we require for any $1 \leq i \leq n$, the set of free variables of t_i, $\mathsf{FV}(t_i) \subseteq dom(\mu) = \{x_1, ..., x_n\}$ and we also do not allow reduction, definition substitution, substitution inside the closure, we call it *local property*, without this property, we are in the dangerous of losing confluence property(see [2] for a detailed discussion). $\mu \in t$ means the closure μ appears in t. $\vec{\mu}t$ denotes $\mu_1...\mu_n t$. $[t'/x](\mu t) \ \equiv \ \mu([t'/x]t)$. So $\mathsf{FV}(\mu t) = \mathsf{FV}(t) - dom(\mu)$.

Definition 22 (Beta-Reductions)**.**

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

Definition 23 (Mu-Reductions)**.**

$$\frac{dom(\mu)\#\mathsf{FV}(t)}{\mu t \to_\mu t} \quad \frac{}{\mu(\lambda x.t) \to_\mu \lambda x.\mu t} \quad \frac{}{\mu(t_1 t_2) \to_\mu (\mu t_1)(\mu t_2)} \quad \frac{t \to_\mu t'}{\lambda x.t \to_\mu \lambda x.t'}$$

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

5.2.2 Confluence of Local λ_μ Calculus

Lemma 16. *$\to_\mu$ is strongly normalizing and confluent.*

Definition 24 (μ-Normal Forms)**.**
$n \ ::= \ x \mid \mu x_i \mid \lambda x.n \mid nn'$

We require $x_i \in dom(\mu)$.

Definition 25 (μ-Normalize Funciton)**.**

$m(x) \ := \ x$ $\qquad m(\lambda y.t) \ := \ \lambda y.m(t)$
$m(t_1 t_2) \ := \ m(t_1)m(t_2)$ $\qquad m(\vec{\mu}y) \ := y \text{ if } y \notin dom(\vec{\mu})$.
$m(\vec{\mu}y) \ := \mu_i y \text{ if } y \in dom(\mu_i)$. $\qquad m(\vec{\mu}(tt')) \ := m(\vec{\mu}t)m(\vec{\mu}t')$
$m(\vec{\mu}(\lambda x.t)) \ := \lambda x.m(\vec{\mu}t)$.

Lemma 17. *Let Φ denote the set of μ normal form, for any term t, $m(t) \in \Phi$.*

Definition 26 (β Reduction on μ-normal Forms)**.**

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

Note that the last three rules follows from the first rule. For the second one, because $n \to_\beta t$ implies $\lambda x.n \to_\beta \lambda x.t$ and $m(\lambda x.t) \equiv \lambda x.m(t)$. The others follow similarly.

Definition 27 (Parallelization)**.**

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

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

Lemma 18. $\to_{\beta\mu} \subseteq \Rightarrow_{\beta\mu} \subseteq \to_{\beta\mu}^*$.

Lemma 19. *If $n_2 \Rightarrow_{\beta\mu} n_2'$, then $m([n_2/x]n_1) \Rightarrow_{\beta\mu} m([n_2'/x]n_1)$.*

Lemma 20. *$m(m(t)) \equiv m(t)$ and $m([m(t_1)/y]m(t_2)) \equiv m([t_1/y]t_2)$.*

Lemma 21. *If $n_1 \Rightarrow_{\beta\mu} n_1'$ and $n_2 \Rightarrow_{\beta\mu} n_2'$, then $m([n_2/x]n_1) \Rightarrow_{\beta\mu} m([n_2'/x]n_1')$.*

Lemma 22. *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.*

One can also use Takahashi's method to prove the lemma above. We will not explore that here.

Lemma 23. *If $a \to_\beta b$, then $m(a) \to_{\beta\mu}^* m(b)$.*

Theorem 4. *$\to_\beta \cup \to_\mu$ is confluent.*

Proof. By lemma 15. ∎

5.3 Type Preservation and Confluence

Recall the statement of the type preservation: If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$. In dependent type system, the following conversion rule is presented:

$$\frac{\Gamma \vdash t : T' \quad T = T'}{\Gamma \vdash t : T} \; Conv$$

The common method to prove type preservation is by induction on the derivation of $\Gamma \vdash t : T$. One will reach the case of $\Gamma \vdash (\lambda x.t_1)t_2 : T$, where $\Gamma \vdash \lambda x.t_1 : T_1 \to T_2$, $\Gamma \vdash t_2 : T_1$ and $T_2 = T$. Since $(\lambda x.t_1)t_2 \to [t_2/x]t_1$, we need to show $\Gamma \vdash [t_2/x]t_1 : T$. $\Gamma \vdash \lambda x.t_1 : T_1 \to T_2$ implies $\Gamma, x : T_1' \vdash t_1 : T_2'$ and $T_1' \to T_2' = T_1 \to T_2$. It would be desirable to have $T_1' = T_1$ and $T_2' = T_2$, then we would have $\Gamma, x : T_1' \vdash t_1 : T_2$ and $\Gamma \vdash t_2 : T_1'$, so we should be able to get $\Gamma \vdash [t_2/x]t_1 : T_2$ and $T_2 = T$.

So the question is: given that $T_1' \to T_2' = T_1 \to T_2$, is it true that $T_1' = T_1$, $T_2' = T_2$? We called this *inverse structure congruence* problem. We know that given $T_1' = T_1$, $T_2' = T_2$, one can conclude that $T_1' \to T_2' = T_1 \to T_2$. It is not immediate that the inverse structure congruence holds, so we need to analyze the convertability relation between $T_1' \to T_2'$ and $T_1 \to T_2$.
We would like the following invertability property holds.

Definition 28 (Inverse Structure Congruence)**.** *$T_1 \to T_2 = T_1' \to T_2'$ implies $T_1 = T_1'$ and $T_2 = T_2'$.*

A reducton system $(\mathcal{T}, \rightarrowtail)$, where $\mathcal{T}$ is a set of types, arise when we analyze the relation $T = T'$. Often for such system we want to design $\rightarrowtail$ such that $T_1 \to T_2$ can only be reduced to $T_1' \to T_2'$ when $T_1 \rightarrowtail T_1'$ or $T_2 \rightarrowtail T_2'$. So confluence of $(\mathcal{T}, \rightarrowtail)$ will imply that, for $T_1 \to T_2 = T_1' \to T_2'$, there is a T_3 such that $T_1 \to T_2 \stackrel{*}{\rightarrowtail} T_3$ and $T_1' \to T_2' \stackrel{*}{\rightarrowtail} T_3$. So we know T_3 must be of the form $T_4 \to T_5$. So $T_1 \stackrel{*}{\rightarrowtail} T_4 \stackrel{*}{\leftarrowtail} T_1'$ and $T_2 \stackrel{*}{\rightarrowtail} T_5 \stackrel{*}{\leftarrowtail} T_2'$, thus $T_1 = T_1'$ and $T_2 = T_2'$. So that is why confluence can be used to get the inverse structure congruence property, thus to prove type preservation. This machinery can be better illustrated by example, namely, the proof of type preservation for $\mathsf{Selfstar}$, but we will have to leave that to future work.

6 Future Explorations and Conclusions

6.1 System $\mathsf{Selfstar}$

This section we present a novel type system that extends dependent type system with recursive definitions, $* : *$ and self type. The type for Church numerals and Scott numerals reflect a form of induction principle. As a dependent typed programming language, we think it provides an alternative design approach handle data types in functional programming language. While due to the present of recursive definition and $* : *$, we do not have Curry-Howard correspondent in this type system, so terms in $\mathsf{Selfstar}$ does not correspond to proofs in intuitionistic logic.

Definition 29.
Term t $::= \; * \mid x \mid \lambda x.t \mid tt' \mid \mu t \mid \Pi x : t_1.t_2 \mid \iota x.t.$
Closure μ $::= \{x_i \mapsto t_i\}_{i \in \mathcal{I}}$
Context Γ $::= \; \cdot \mid \Gamma, x : t \mid \Gamma, \tilde{\mu}$

We called $\iota x.t$ self type and the closure μ is used for mutually recusive definitions, it follows the same convention in section 5.2.1. $\tilde{\ }$ is an operation(we call it *lifting*). If μ is $\{x_i \mapsto t_i\}_{i \in \mathcal{I}}$, then $\tilde{\mu}$ is $\{(x_i : a_i) \mapsto t_i\}_{i \in \mathcal{I}}$.

We collapse the syntax of terms and types, so the notion of types only arise when we have the judgement $\Gamma \vdash t : t'$, we call t' the type of t. We list only some essential rules for typing.

Definition 30 (Typing)**.**

$$\frac{\Gamma, x : t_1 \vdash t_2 : * \quad \Gamma \vdash t_1 : *}{\Gamma \vdash \Pi x : t_1.t_2 : *}\ \textit{Pi} \qquad \frac{(x : t) \in \Gamma}{\Gamma \vdash x : t}\ \textit{Var} \qquad \frac{}{\Gamma \vdash * : *}\ \textit{Star}$$

$$\frac{\Gamma \vdash t : \Pi x : t_1.t_2 \quad \Gamma \vdash t' : t_1}{\Gamma \vdash tt' : [t'/x]t_2}\ \textit{App} \qquad \frac{\Gamma \vdash t : \iota x.t'}{\Gamma \vdash t : [t/x]t'}\ \textit{SelfInst} \qquad \frac{\Gamma \vdash t : [t/x]t'}{\Gamma \vdash t : \iota x.t'}\ \textit{SelfGen}$$

$$\frac{\Gamma \vdash t : t_1 \quad \Gamma \vdash t_1 = t_2}{\Gamma \vdash t : t_2}\ \textit{Conv} \qquad \frac{\Gamma, x : t_1 \vdash t : t_2 \quad \Gamma \vdash t_1 : *}{\Gamma \vdash \lambda x.t : \Pi x : t_1.t_2}\ \textit{Lam} \qquad \frac{\Gamma, x : \iota x.t \vdash t : *}{\Gamma \vdash \iota x.t : *}\ \textit{Self}$$

$$\frac{\Gamma, \tilde{\mu} \vdash t : t' \quad \{\Gamma, \tilde{\mu} \vdash t_j : a_j\}_{(t_j : a_j) \in \tilde{\mu}}}{\Gamma \vdash \mu t : \mu t'}\ \textit{Mu}$$

For type $\Pi x : t_1.t_2$ if the variable x does not appear in t_2, we write $t_1 \to t_2$ instead. Note that $\to$ in this section has nothing to do with reduction. Now we can see how to type Church encoding and Scott encoding with self type and resursvie defintion.

Definition 31 (Church Encoding)**.** *Let $\tilde{\mu_c}$ be the following recursive defintions:*
$(\mathsf{Nat} : *) \mapsto \iota x.\Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ x)$
$(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z)$
$(0 : \mathsf{Nat}) \mapsto \lambda C.\lambda s.\lambda z.z$

Now let us see how we can derive $\tilde{\mu_c} \vdash \lambda C.\lambda s.\lambda z.z : \mathsf{Nat}$ and $\tilde{\mu_c} \vdash \lambda n.\lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z) : \mathsf{Nat} \to \mathsf{Nat}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\tilde{\mu_c}, C : \mathsf{Nat} \to *, s : (\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))), z : C\ 0 \vdash z : C\ 0}\ Var}{\tilde{\mu_c}, C : \mathsf{Nat} \to *, s : (\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))), z : C\ 0 \vdash z : C\ (\lambda C.\lambda s.\lambda z.z)}\ Conv}{\tilde{\mu_c} \vdash \lambda C.\lambda s.\lambda z.z : \Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ (\lambda C.\lambda s.\lambda z.z))}\ Lam}{\tilde{\mu_c} \vdash \lambda C.\lambda s.\lambda z.z : \iota x.\Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ x)}\ SelfGen}{\tilde{\mu_c} \vdash \lambda C.\lambda s.\lambda z.z : \mathsf{Nat}}\ Conv$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta_1}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash s\ n : (C\ n) \to (C\ (\mathsf{S}\ n))}\ App \quad \dfrac{\Delta_2}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash n\ C\ s\ z : C\ n}\ App}{\tilde{\mu_c}, n : \mathsf{Nat}, C : \mathsf{Nat} \to *, s : (\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))), z : C\ 0 \vdash s\ n\ (n\ C\ s\ z) : C\ (\mathsf{S}\ n)}\ App}{\tilde{\mu_c}, n : \mathsf{Nat} \vdash \lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z) : \Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ (\mathsf{S}\ n))}\ Lam}{\tilde{\mu_c}, n : \mathsf{Nat} \vdash \lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z) : \iota x.\Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ x)}\ =\iota}{\tilde{\mu_c}, n : \mathsf{Nat} \vdash \lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z) : \mathsf{Nat}}}{\tilde{\mu_c} \vdash \lambda n.\lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z) : \mathsf{Nat} \to \mathsf{Nat}}\ Lam$$

In above derivation, $\Gamma = C : \mathsf{Nat} \to *, s : (\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))), z : C\ 0$, and $=\iota$ step first convert $\mathsf{S}\ n$ to $\lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z)$, then apply the SelfGen rule. The Δ_1 is a subderivation:

$$\dfrac{\dfrac{}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash s : \Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))}\ Var \quad \dfrac{}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash n : \mathsf{Nat}}\ Var}{\Delta_1}$$

The Δ_2 is a subderivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash n : \mathsf{Nat}}\ Var}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash n : \iota x.\Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ x)}\ Conv}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash n : \Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ n)}\ SelfInst \quad \Delta_3}{\Delta_2}$$

where Δ_3:

$$\dfrac{\dfrac{}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash C : \mathsf{Nat} \to *}\ Var \quad \dfrac{}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash s : \Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))}\ Var \quad \dfrac{}{\tilde{\mu_c}, n : \mathsf{Nat}, \Gamma \vdash z : C\ 0}\ Var}{\Delta_3}$$

The derivations above are a little lengthy, we present that only for the purpose of demonstration, we will not give any derivation any more. Now the induction principle for Church encoding can be expressed as: $\tilde{\mu_c} \vdash (\mathsf{Ind}\ := \lambda C.\lambda s.\lambda z.\lambda n.n\ C\ s\ z) : \Pi C : \mathsf{Nat} \to *.\Pi n : \mathsf{Nat}.((C\ n) \to (C\ (\mathsf{S}\ n))) \to C\ 0 \to \Pi n : \mathsf{Nat}.C\ n$.

With $* : *$, we now can define Leibniz's equality as $\mathsf{Eq}\ := \lambda A.\lambda x.\lambda y.\Pi C : (A \to *).C\ x \to C\ y$. Now we have the judgement $\cdot \vdash \mathsf{Eq} : \Pi A : *.A \to A \to *$. Now define addition:
$\tilde{\mu_c} \vdash (\mathsf{add} := \lambda n.\lambda m.\mathsf{Ind}\ (\lambda y.\mathsf{Nat})\ (\lambda x.\mathsf{S})\ m\ n) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
Now one can use induction principle to derive $\tilde{\mu_c} \vdash t : \Pi m : \mathsf{Nat}.(\mathsf{Eq}\ \mathsf{Nat}\ (\mathsf{add}\ m\ 0)\ m)$ for some term t, the term t is a lambda expression that encode a proof of the formula $\Pi m : \mathsf{Nat}.(\mathsf{Eq}\ \mathsf{Nat}\ (\mathsf{add}\ m\ 0)\ m)$ (Namely, by induction on natural number m). Recalled that in this system we do not have Curry-Howard correspondent, so t does not in general corresponds to a proofs of its type, but for the derivation of $\tilde{\mu_c} \vdash t : \Pi m : \mathsf{Nat}.(\mathsf{Eq}\ \mathsf{Nat}\ (\mathsf{add}\ m\ 0)\ m)$, we do not use any illogical principle, we still want to say it is a valid proof. So for future work, we want to identify a fragment of Selfstar that is logically consistent.

Definition 32 (Scott Encoding)**.** *Let $\tilde{\mu_s}$ be the following recursive defintions:*
$(\mathsf{Nat} : *) \mapsto \iota x.\Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.C\ (\mathsf{S}\ n)) \to (C\ 0) \to (C\ x)$
$(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda C.\lambda s.\lambda z.s\ n$
$(0 : \mathsf{Nat}) \mapsto \lambda C.\lambda s.\lambda z.z$

With Scott numerals defined above, one can derive a case analysis principle:
$\tilde{\mu_s} \vdash (\mathsf{Case}\ := \lambda C.\lambda s.\lambda z.\lambda n.n\ C\ s\ z) : \Pi C : \mathsf{Nat} \to *.\Pi n : \mathsf{Nat}.(C\ (\mathsf{S}\ n)) \to C\ 0 \to \Pi n : \mathsf{Nat}.C\ n$
addition function can also be defined by extending the closure $\tilde{\mu_s}$ by
$(\mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda m.\mathsf{Case}\ (\lambda n.\mathsf{Nat})\ \ (\lambda p.(\mathsf{S}\ (\mathsf{add}\ p\ m)))\ m\ n$

One can further prove theorems about the add function like what we did for Church encoding, we will not pursue that here. Interestingly the expression for Case and Ind are the same, they are used in the add operation for the typing purpose. Comparing with Church version, one can see two styles of defining addition, one through iteration, the other through recurison, both of which are expressible within the Selfstar system.

6.2 Conclusion and Future Works

Conclusion: We present two methods to represent natural number as lambda terms, namely, Church encoding and Scott encoding. We also surveyed type systems from simply typed lambda calculus to second order dependent type systems. Church encoding with system **F** and Scott encoding with recursive type are discussed. Some of the problems with Church encoding data in dependent type systems are addressed. System Selfstar is presented as a respond to the problems arise in dependent type system, and also to general data type design in functional programming language.

Type preservation problem of Selfstar leads us to another line of works that related to term rewriting. The notion of abstract reduction system is introduced, and several methods to prove confluence are included. The a fragment of term system of the Selfstar is shown to be confluent. The connection between confluence and type preservation is illustrated. Church and Scott encoding numerals are typed in Selfstar, together with the corresponding induction and case analysis principles, some simple theorems are presented to demonstrate the logical reasoning.

Future Works: We want to extend the confluence of the λ_μ to the whole Selfstar system, then to establish type preservation. We also want to identify a logical fragment of Selfstar and show this logical fragment is consistent. Last but not least, we want to refine the prototype system to reflects some of the new ideas from the analysis of Selfstar.

References

[1] M. Abadi, L. Cardelli, and G. Plotkin. Types for the Scott Numerals, 1993. Unpublished, available from Abadi's web page.

[2] Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154 – 233, 1997.

[3] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2005.

[4] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.

[5] Hendrik Pieter Barendregt. *The lambda calculus: Its syntax and semantics*, volume 103. North Holland, 1985.

[6] Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[7] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[8] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. Term rewriting systems. terese. 2003.

[9] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[10] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies).* Princeton University Press, Princeton, NJ, USA, 1985.

[11] T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, September 1989.

[12] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.

[13] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88*, pages 50–66. Springer, 1990.

[14] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, 1996.

[15] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II.* North-Holland, 1972.

[16] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities.* Addison-Wesley Professional, 2006.

[17] Peter Forrest. The identity of indiscernibles. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Winter 2012 edition, 2012.

[18] Gottlob Frege. The basic laws of arithmetic: Exposition of the system, translated and edited with an introduction by montgomery furth, 1967.

[19] Gerhard Gentzen. Investigations into logical deduction. *American philosophical quarterly*, 1(4):288–306, 1964.

[20] Herman Geuvers. Induction is not derivable in second order dependent type theory. In *Proceedings of the 5th international conference on Typed lambda calculi and applications*, TLCA'01, pages 166–181, Berlin, Heidelberg, 2001. Springer-Verlag.

[21] Eduardo Giménez and Pierre Castéran. A tutorial on [co-] inductive types in coq, 2005.

[22] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, 1972.

[23] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* Cambridge University Press, New York, NY, USA, 1989.

[24] Benjamin Grégoire and Jorge Luis Sacchini. On strong normalization of the calculus of constructions with type-based termination. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 333–347. Springer, 2010.

[25] Thérèse Hardin. Confluence results for the pure strong categorical logic ccl. λ-calculi as subsystems of ccl. *Theor. Comput. Sci.*, 65(3):291–342, July 1989.

[26] A. Heyting. *Die formalen Regeln der intuitionistischen Logik.* Sitzungsberichte der Preussischen Akademie der Wissenschaften. Physikalisch-mathematische Klasse. Deütsche Akademie der Wissenschaften zu Berlin, Mathematisch-Naturwissenschaftliche Klasse, 1930.

[27] J Roger Hindley. *Basic simple type theory*, volume 42. Cambridge University Press, 1997.

[28] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[29] J.M. Jansen, R. Plasmeijer, and P. Koopman. Functional pearl: Comprehensive encoding of data types and algorithms in the lambda-calculus. *Internal report, NLDA*, 2011.

[30] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples,, Italy, 1984.

[31] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer Berlin Heidelberg, 2005.

[32] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The definition of standard ml, revised edition. *MIT Press*, 1(2):2–3, 1997.

[33] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *Typed lambda calculi and applications*, pages 328–345. Springer, 1993.

[34] Giuseppe Peano. *Arithmetices principia: nova methodo.* Fratres Bocca, 1889.

[35] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. *ACM SIGPLAN Notices*, 23(7):199–208, 1988.

[36] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[37] Aaron Stump. Directly reflective meta-programming. *Higher Order Symbol. Comput.*, 22(2):115–144, June 2009.

[38] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in guru. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, pages 49–58, New York, NY, USA, 2008. ACM.

[39] Masako Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1):120–127, 1995.

[40] The Coq Development Team. The coq proof assistant reference manual. *Version 8.3. INRIA*, 2010.

[41] Myra VanInwegen. *The machine-assisted proof of programming language properties.* PhD thesis, University of Pennsylvania, 1996.

[42] John Von Neumann. First draft of a report on the edvac. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993.

[43] 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. The TYPES 1992 proceedings are available at `http://www.cse.chalmers.se/research/group/logic/Types/proc92.ps`.

[44] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[45] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

A Proofs

A.1 Proof of Lemma 17

Let Φ denote the set of μ normal form, for any term t, $m(t) \in \Phi$.

Proof. One way to prove this is first identify t as $\vec{\mu_1} t'$, here $\vec{\mu_1}$ means there are zero or more closures and t' does not contains any closure at head position. Then we can proceed by induction on the structure of t':

Base Cases: $t' = x$, obvious.

Step Cases: If $t' = \lambda x.t''$, then $m(\vec{\mu_1}(\lambda x.t'')) \equiv \lambda x.m(\vec{\mu_1} t'')$. Now we can again identify t'' as $\vec{\mu_2} t'''$, where t''' does not have any closure at head position. Since t''' is structurally smaller than $\lambda x.t''$, by IH, $m(\vec{\mu_1}\vec{\mu_2} t''') \in \Phi$, thus $m(\vec{\mu_1}(\lambda x.t'')) \equiv \lambda x.m(\vec{\mu_1} t'') \in \Phi$.

For $t' = t_1 t_2$, we can argue similarly as above.

□

A.2 Proof of Lemma 19

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

Proof. By induction on the structure of n_1. We list a few non-trivial cases:

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

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

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

□

A.3 Proof of Lemma 20

$m(m(t)) \equiv m(t)$ and $m([m(t_1)/y]m(t_2)) \equiv m([t_1/y]t_2)$.

Proof. The first equality is by lemma 17. For the second equality, we prove it through similar method as lemma 17: We identify t_2 as $\overrightarrow{\mu_1}t_2'$, 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 $m(m(t)) \equiv m(t)$.

Step Cases: If $t_2' = \lambda x.t_2''$, then $m(\overrightarrow{\mu_1}(\lambda x.[t_1/y]t_2'')) \equiv \lambda x.m(\overrightarrow{\mu_1}([t_1/y]t_2'')) \equiv \lambda x.m(\overrightarrow{\mu_1}\overrightarrow{\mu_2}([t_1/y]t_2'''))$, where t_2'' as $\overrightarrow{\mu_2}t_2'''$ and t_2''' does not have any closure at head position. Since t_2''' is structurally smaller than $\lambda x.t_2''$, by IH, $m(\overrightarrow{\mu_1}\overrightarrow{\mu_2}([t_1/y]t_2''')) \equiv m([t_1/y](\overrightarrow{\mu_1}\overrightarrow{\mu_2}t_2''')) \equiv m([m(t_1)/y]m(\overrightarrow{\mu_1}\overrightarrow{\mu_2}t_2'''))$. Thus $\lambda x.m(\overrightarrow{\mu_1}\overrightarrow{\mu_2}([t_1/y]t_2''')) \equiv \lambda x.m([m(t_1)/y]m(\overrightarrow{\mu_1}\overrightarrow{\mu_2}t_2'''))$. So $m([t_1/y]\overrightarrow{\mu_1}(\lambda x.t_2'')) \equiv m([m(t_1)/y]m(\lambda x.\overrightarrow{\mu_1}\overrightarrow{\mu_2}t_2''')) \equiv m([m(t_1)/y]m(\lambda x.\overrightarrow{\mu_1}t_2'')) \equiv m([m(t_1)/y]m(\overrightarrow{\mu_1}(\lambda x.t_2'')))$

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

□

A.4 Proof of Lemma 21

If $n_1 \Rightarrow_{\beta\mu} n_1'$ and $n_2 \Rightarrow_{\beta\mu} n_2'$, then $m([n_2/y]n_1) \Rightarrow_{\beta\mu} m([n_2'/y]n_1')$.

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

Base Case:

$$\frac{}{n \Rightarrow_{\beta\mu} n}$$

By the lemma 19.

Base Case:

$$\frac{x_i \mapsto t_i \in \mu}{\mu x_i \Rightarrow_{\beta\mu} m(\mu t_i)}$$

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

Step Case:

$$\frac{n_a \Rightarrow_{\beta\mu} n_a' \quad n_b \Rightarrow_{\beta\mu} n_b'}{(\lambda x.n_a)n_b \Rightarrow_{\beta\mu} m([n_a'/x]n_b')}$$

We have $m((\lambda x.[n_2/y]n_a)[n_2/y]n_b) \equiv (\lambda x.m([n_2/y]n_a))m([n_2/y]n_b)$
$\overset{IH}{\Rightarrow}_{\beta\mu} m([m([n_2'/y]n_b')/x]m([n_2'/y]n_a')) \equiv m([n_2'/y]([n_b'/x]n_a'))$. The last equality is by lemma 20.

Step Case:

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

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

Step Case:

$$\frac{n_a \Rightarrow_{\beta\mu} n'_a \quad n_b \Rightarrow_{\beta\mu} n'_b}{n_a n_b \Rightarrow_{\beta\mu} n'_a n'_b}$$

We have $m([n_2/y]n_a[n_2/y]n_b) \equiv m([n_2/y]n_a)m([n_2/y]n_b)$
$\overset{IH}{\Rightarrow}_{\beta\mu} m([n'_2/y]n'_a)m([n'_2/y]n'_b) \equiv m([n'_2/y](n'_a n'_b))$.

□

A.5 Proof of Lemma 22

If $n \Rightarrow_{\beta\mu} n'$ and $n \Rightarrow_{\beta\mu} n''$, then there exist n''' such that $n'' \Rightarrow_{\beta\mu} n'''$ and $n' \Rightarrow_{\beta\mu} n'''$.

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

$$\frac{}{n \Rightarrow_{\beta\mu} n}$$

Obvious.

Base Case:

$$\frac{}{\mu x_i \Rightarrow_{\beta\mu} m(\mu t_i)}$$

Obvious.

Step Case:

$$\frac{n_1 \Rightarrow_{\beta\mu} n'_1 \quad n_2 \Rightarrow_{\beta\mu} n'_2}{(\lambda x.n_1)n_2 \Rightarrow_{\beta\mu} m([n'_1/x]n'_2)}$$

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 21 and IH, we have $m([n'_1/x]n'_2) \Rightarrow_{\beta\mu} m([n'''_1/x]n'''_2)$. We also have $(\lambda x.n''_1)n''_2 \Rightarrow_{\beta\mu} m([n'''_1/x]n'''_2)$, 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 $(\lambda x.n_1)n_2 \Rightarrow_{\beta\mu} m([n''_2/x]n''_1)$, where $n_1 \Rightarrow_{\beta\mu} n''_1$ and $n_2 \Rightarrow_{\beta\mu} n''_2$. By lemma 21 and IH, we have $m([n'_1/x]n'_2) \Rightarrow_{\beta\mu} m([n'''_1/x]n'''_2)$ and $m([n''_1/x]n''_2) \Rightarrow_{\beta\mu} m([n'''_1/x]n'''_2)$.

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

□

A.6 Proof of Lemma 23

Lemma 24. $m(\vec{\mu}\vec{\mu}t) \equiv m(\vec{\mu}t)$ *and* $m(\vec{\mu}([t_2/x]t_1)) \equiv m([\vec{\mu}t_2/x]\vec{\mu}t_1)$

Proof. We can prove this using the same method as lemma 17. We will not prove it here. □

Proposition 1. *If $a \to_\beta b$, then $m(a) \to_{\beta\mu} m(b)$.*

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

Base Case:

$$\frac{(x_i \mapsto t_i) \in \mu}{\mu x_i \to_\beta \mu t_i}$$

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

Base Case:

$$\frac{}{(\lambda x.t)t' \to_\beta [t'/x]t}$$

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

Step Case:

$$\frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'}$$

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

Step Case:

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

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

If $dom(\mu) \cap FV(t) \neq \emptyset$, then identify t as $\dot{\vec{\mu}}_1 t''$, where t'' does not contain any closure at head position. We do case analyze on the structure of t'':

Case. $t'' = x_i \in dom(\dot{\vec{\mu}}_1)$ or $x_i \notin dom(\dot{\vec{\mu}}_1)$, these cases will not arise.

Case. $t'' = \lambda y.t_1$, then it must be that $t' = \dot{\vec{\mu}}_1(\lambda y.t_1')$ where $t_1 \to_\beta t_1'$. So we get $\mu\dot{\vec{\mu}}_1 t_1 \to_\beta \mu\dot{\vec{\mu}}_1 t_1'$. By IH(depth of $\mu\dot{\vec{\mu}}_1 t_1 \to_\beta \mu\dot{\vec{\mu}}_1 t_1'$ is smaller), we have $m(\mu\dot{\vec{\mu}}_1 t_1) \to_{\beta\mu} m(\mu\dot{\vec{\mu}}_1 t_1')$. Thus $m(\mu\dot{\vec{\mu}}_1(\lambda y.t_1)) \equiv \lambda y.m(\mu\dot{\vec{\mu}}_1 t_1) \to_{\beta\mu} \lambda y.m(\mu\dot{\vec{\mu}}_1 t_1') \equiv m(\mu\dot{\vec{\mu}}_1(\lambda y.t_1'))$.

Case. $t'' = t_1 t_2$ and $t' = \dot{\vec{\mu}}_1(t_1' t_2)$, where $t_1 \to_\beta t_1'$. We have $\mu\dot{\vec{\mu}}_1 t_1 \to_\beta \mu\dot{\vec{\mu}}_1 t_1'$. By IH(depth of $\mu\dot{\vec{\mu}}_1 t_1 \to_\beta \mu\dot{\vec{\mu}}_1 t_1'$ is smaller), $m(\mu\dot{\vec{\mu}}_1 t_1) \to_{\beta\mu} m(\mu\dot{\vec{\mu}}_1 t_1')$. Thus $m(\mu\dot{\vec{\mu}}_1(t_1 t_2)) \equiv m(\mu\dot{\vec{\mu}}_1 t_1)m(\mu\dot{\vec{\mu}}_1 t_2) \to_{\beta\mu} m(\mu\dot{\vec{\mu}}_1 t_1')m(\mu\dot{\vec{\mu}}_1 t_2) \equiv m(\mu\dot{\vec{\mu}}_1(t_1' t_2))$. 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 $t' = \dot{\vec{\mu}}_1([t_2/y]t_1)$. $m(\mu\dot{\vec{\mu}}_1((\lambda y.t_1)t_2)) \equiv (\lambda y.m(\mu\dot{\vec{\mu}}_1 t_1))m(\mu\dot{\vec{\mu}}_1 t_2) \to_{\beta\mu} m([m(\mu\dot{\vec{\mu}}_1 t_2)/y]m(\mu\dot{\vec{\mu}}_1 t_1)) \equiv m([\mu\dot{\vec{\mu}}_1 t_2/y]\mu\dot{\vec{\mu}}_1 t_1) \equiv m(\mu\dot{\vec{\mu}}_1[t_2/y]t_1)$(lemma 24).

$\square$