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 distinguishes 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 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 \(\text{Plus}(2,1) = 3\). Though this idea of data as program is familiar 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, \((\lambda x. f\ x)\ a \rightarrow \beta f\ a\). (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 whether these assumptions are met when composing them together. For example, when a programmer write a function with type \(\text{int} \rightarrow \text{int}\), 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 inferred as $\text{int} \rightarrow \text{int}$. 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, $\text{In} : \text{Num} . \text{Array} (\text{String}, n)$), 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 $\text{int}$. 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 calculus, 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 limited 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 definition 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 facilities 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 $\text{F}$ [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 inefficient to define certain operation on datatype, 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 $\text{T}$ (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 language, Scott encoding seems to be a better fit than Church encoding [29]. Scott encoding was claimed to be typable in System $\text{F}$ [1], but it is unclear how to type recursive functions on such encoding in System $\text{F}$.

We propose a novel type system called $\text{Selfstar}$, 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 Overview

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. Section 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 \( (\mathcal{A}, \{\to_i\}_{i \in I}) \), where \( \mathcal{A} \) is a set and \( \to_i \) is a binary relation(called reduction) on \( \mathcal{A} \) indexed by a finite nonempty set \( I \).

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 \( (\mathcal{A}, \{\to_i\}_{i \in I}) \), the reflexive transitive closure of \( \to_i \) is written as \( \to^*_i \) or \( \to^*_i \) is defined by:

- \( m \to^*_i m \).
- \( m \to^*_i n \) if \( m \to_i n \).
- \( m \to^*_i l \) if \( m \to_i n, n \to_i l \).

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

- \( m =_i n \) if \( m \to_i n \).
- \( n =_i m \) 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 \( a \to_i b \) and \( b \) is not reducible. \( a \) and \( b \) are joinable if there is \( c \) such that \( a \to_i c \) and \( b \to_i c \). An abstract reduction system is strongly normalizing if there are no infinite reduction path.

2.2 Lambda Calculus

We use \( x, y, z, s, n, x_1, x_2, \ldots \) to denote individual variable, \( t, t', a, b, t_1, t_2, \ldots \) to denote term, \( = \) 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).

\[ \text{Term } t ::= x \mid \lambda x.t \mid t \ t' \]

\[ \text{Reduction } (\lambda x.t)t' \to_\beta [t'/x]t \]
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 \(\lambda\) 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 \(\alpha\)-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 \(\lambda z. \lambda x. z x\).

\((\lambda x. \lambda y. x y)(\lambda z. z) z_1\) \(\to_\beta (\lambda x. \lambda y. x y) z_1 \to_\beta \lambda y. z_1 y\) 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 introduction on lambda calculus, we refer to [5].

2.3 Simple Types

We use \(A, B, C, X, Y, Z, \ldots\) to denote type variable, \(T, S, U, \ldots\) to denote any type.

Definition 6.

\(\text{Type } T ::= X | T_1 \to T_2\)

\(\text{Context } \Gamma ::= \cdot | \Gamma, x : T\)

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 \(\Gamma\), denoted by \(\Gamma \vdash t : T\). We present simply typed lambda calculus below.

Definition 7.

\[\begin{align*}
(x : T) & \in \Gamma \quad \text{Var} \\
\Gamma \vdash x : T & \quad \text{Var} \\
\Gamma \vdash \lambda x. t : T_1 \to T_2 & \quad \text{Abs} \\
\Gamma \vdash t : T_1 \to T_2 \quad \Gamma \vdash t' : T_2 & \quad \text{App}
\end{align*}\]

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 theorems 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).

\(0 ::= \lambda s. \lambda z. z\)

\(S ::= \lambda n. \lambda s. \lambda z. (n s z)\)

From above we know \(1 ::= (\lambda n. \lambda s. \lambda z. (n s z))(\lambda s. \lambda z. z) \to_\beta \lambda s. \lambda z. ((\lambda s. \lambda z. z) s) z =_\beta \lambda s. \lambda z. s z\). Note that the last part of above reductions occur underneath the lambda abstractions. Similarly we can get \(2 ::= \lambda s. \lambda z. s z\).

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 \(\text{it } n f t ::= n f t\). So \(\text{it } 0 f t =_\beta t\) and \(\text{it } (S u) f t =_\beta f(\text{it } u f t)\). So now we can use iterator to define \(\text{Plus } n m ::= \text{it } n S m\).
3.2 Scott Encoding

**Definition 9 (Scott Numeral).**

\[ 0 := \lambda s.\lambda z.z \]
\[ S := \lambda n.\lambda s.\lambda z.s\ n \]

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** \( \text{Fix} := \lambda f.((\lambda x.f\ (x\ x)))((\lambda x.f\ (x\ x))) \). 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 \( \text{Fix} \ g \rightarrow _\beta ((\lambda x.g\ (x\ x)))((\lambda x.g\ (x\ x))) \rightarrow _\beta g((\lambda x.g\ (x\ x)))((\lambda x.g\ (x\ x))) = _\beta g(\text{Fix} \ g) \).

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

So we get \( \text{Rec} \ 0\ f\ v \rightarrow _\beta v \) and \( \text{Rec} \ (S\ n)\ f\ v \rightarrow _\beta f\ ((\text{Rec} \ n\ f\ v)\ n) \). In a similar fashion, one can define \( \text{Plus} : n\ m := \text{Rec} \ (\lambda x.\lambda y.S\ x)\ m \).

The predecessor function can be easily defined as \( \text{Pred} := \text{Rec} \ (\lambda x.\lambda y.y)\ 0 \). It only takes constant time (w.r.t. the number of beta reduction steps) to calculate the predecessor. But this function is tricky to define with Church encoding, one need to first define recursor with iterator, then use recursor to define \( \text{Pred} \).

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:

\[
\frac{\Gamma \vdash t : T \quad X \notin \text{FV} (\Gamma)}{\Gamma \vdash t : \forall X.T} \quad \text{Gen} \quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t : [T/X]T} \quad \text{Inst}
\]

\( X \notin \text{FV} (\Gamma) \) means \( X \) is not a free type variable in the types of the typing context \( \Gamma \). For example, given above typing rule, we can associate identity function with a polymorphic type, i.e. \( \cdot \vdash \lambda x.x : \forall X.X \rightarrow X \).

And we also have \( \cdot \vdash \lambda x.x : T \rightarrow T \) for any type \( T \).

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

\[
\frac{s : X \rightarrow X, z : X}{\cdot \vdash \lambda s.\lambda z.z : (X \rightarrow X) \rightarrow X \rightarrow X}
\]
\[
\frac{s : X \rightarrow X, z : X}{\cdot \vdash \lambda s.\lambda z.z : \forall X.(X \rightarrow X) \rightarrow X \rightarrow X}
\]

For space reason, we only list \( \cdot \vdash 0 : \text{Nat} \), similarly one will can type:

\( \cdot \vdash 5 : \text{Nat} \rightarrow \text{Nat} \)

\( \cdot \vdash \text{It} : \forall X.\text{Nat} \rightarrow (X \rightarrow X) \rightarrow X \rightarrow X \)

\( \cdot \vdash \text{Plus} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \)

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 \( \mu \) is a binder. The additional typing rules are follows:
With recursive type, define $\text{Nat} := \mu X.(X \to U) \to U \to U$ 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 $\text{Nat} \sim (\text{Nat} \to U) \to U \to U$.

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

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 $\mathbf{F}$ with dependent type (product type) $\Pi x : T.T^x$ and index type $T.t$. The additional typing rules are:

- $\frac{\Gamma \vdash x : T'}{\Gamma \vdash \lambda x.t : \Pi x : T'.T}$ \hspace{1cm} $\text{Pi}$
- $\frac{\Gamma \vdash t : \Pi x : T'.T \hspace{1cm} \Gamma \vdash t' : T'}{\Gamma \vdash t : \Pi x : T'.T}$ \hspace{1cm} $\text{Elim}$
- $\frac{\Gamma \vdash t : [t_1/x]T \hspace{1cm} t_1 =_{\beta} t_2}{\Gamma \vdash t : [t_2/x]T}$ \hspace{1cm} $\text{Conv}$

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 $\kappa$) 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 $\mathbf{F}$ to $\forall X : \kappa.T$ to allow a finer classification of type. We say type $T$ has kind $\kappa$ under the context $\Gamma$, denoted by $\Gamma \vdash T : \kappa$.

**Definition 10** (Kind and Kinding). $\text{Kind } \kappa := * \mid \xi x : T.\kappa$

**Context** $\Gamma := \cdot \mid \Gamma, x : T \mid \Gamma, X : \kappa$
which is called the substitution property and the relation \( \text{Eq} \\text{Id} \) that induction is not derivable in second order dependent type system [20]. Note that induction principle associativity etc. For such properties, the proof normally will involve induction argument, but it is known for \( \forall \) formula [7]. One can also derive a proof for term \( \lambda a.\lambda x.x \) means there does not exist any derivation and term \( t \), since under Curry-Howard correspondent, dependent type can be interpreted as intuitionistic logic [17]. Now we can show \( \vdash \lambda x.x : \text{Eq} [\text{Nat}] (\text{Plus} 1 1) 2 \). It is derivable because the following derivation:

\[
\begin{align*}
\text{Conv} & \quad \Delta : (\xi z : \text{Nat}.*) , x : C (\text{Plus} 1 1) \vdash x : C (\text{Plus} 1 1) \\
\text{Var} & \quad \Delta : (\xi z : \text{Nat}.*) , x : C (\text{Plus} 1 1) \vdash x : C \\
\text{Abs} & \quad \vdash \lambda x.x : \forall C : (\xi z : \text{Nat}.*) , C (\text{Plus} 1 1) \rightarrow C 2 \\
\text{Gen} & \quad \vdash \lambda x.x : \text{Eq} [\text{Nat}] (\text{Plus} 1 1) 2
\end{align*}
\]

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

\[
\begin{align*}
\text{Var} & \quad \Delta : * , a : A , C : (\xi z : A.*) , x : C a \vdash x : C a \\
\text{Abs} & \quad \Delta : * , a : A , C : (\xi z : A.*) \vdash \lambda x.x : C a \rightarrow C a \\
\text{Gen} & \quad \Delta : * , a : A \vdash \lambda x.x : \forall C : (\xi z : A.*) , C a \rightarrow C a \\
\text{Def} & \quad \Delta : * , a : A \vdash \lambda a.\lambda x.x : \Pi a : A . \text{Eq} [A] a a \\
\text{Abs} & \quad \Delta : * , a : A \vdash \lambda a.\lambda x.x : \forall A : * , \Pi a : A . \text{Eq} [A] a a \\
\text{Gen} & \quad \Delta : * , a : A \vdash \lambda a.\lambda x.x : \forall A : * , \Pi a : A . \text{Eq} [A] a a
\end{align*}
\]

The derivation above can be viewed as a proof of reflexive for the \( \text{Eq} \)(Namely, the formula \( \forall A : * . \Pi a : A . \text{Eq} [A] a a \); or it can be viewed as a procedure to associate the type \( \forall A : * . \Pi a : A . \text{Eq} [A] a a \) to a lambda term \( \lambda a.\lambda x.x \). Since under Curry-Howard correspondent, dependent type can be interpreted as intuitionistic logic [7]. One can also derive a proof for \( \forall A : * . \Pi a : A . \Pi b : A . \forall B : (\xi x : A.*) . B a \rightarrow (\text{Eq} [A] a b) \rightarrow B b \), 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 \( \text{Plus} \) and the relation \( \text{Eq} \). For functions like \( \text{Plus} \), 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 \( \text{Id} := \forall P : (\xi x : \text{Nat}.*) . P 0 \rightarrow (\Pi y : \text{Nat} . (P y) \rightarrow (P (\text{S} y))) \rightarrow \Pi x : \text{Nat} . P x. \) So this means there does not exist any derivation and term \( t \) such that \( \vdash t : \text{Id} \). Because, informally, we can only have...
Proof. Assume the same notation as definition 12.

An abstract reduction system

Lemma 1. An abstract reduction system \( R \) is confluent iff it is Church-Rosser.

Proof. Assume the same notation as definition 12.

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

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

If \( a \rightarrow b \Rightarrow a = b \), then let \( c \) be \( b \).
If \( b = a \Rightarrow a = b \), by induction, there is a \( c \) such that \( b \rightarrow c \) and \( a \rightarrow c \).

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

\[
\begin{array}{c}
a
\end{array}
\begin{array}{c}
\downarrow c_1
\end{array}
\begin{array}{c}
\downarrow c_2
\end{array}
\begin{array}{c}
c
\end{array}
\begin{array}{c}
d
\end{array}
\begin{array}{c}
\downarrow a
\end{array}
\begin{array}{c}
\downarrow b
\end{array}
\]

The definition of \( = \) depends on \( \rightarrow \), the definition of \( \rightarrow \) depends on \( \rightarrow \), confluence is often easier to prove compare to Church-Rosser, in the sense that it is easier to analyze \( \rightarrow \) 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 different normal forms for \( a \). So we have \( a \rightarrow b \) and \( a \rightarrow c \), by confluence, there exist a \( d \) such that \( b \rightarrow d \) and \( c \rightarrow d \). 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 \( (\mathcal{A}, \{\rightarrow_i\}_{i \in I}) \), 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 \( c \rightarrow d \).

\[
\begin{array}{c}
a
\end{array}
\begin{array}{c}
\downarrow b
\end{array}
\begin{array}{c}
\downarrow c
\end{array}
\begin{array}{c}
d
\end{array}
\]

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

**Proof.** By simple diagram chasing suggested below:
Lemma 3. If exist some $\rightarrow_i$, $\rightarrow_i \subseteq \rightarrow_i \subseteq \rightarrow_i$ and $\rightarrow_i$ satisfies diamond property, then $\rightarrow_i$ is confluent.

Proof. Since $\rightarrow_i \subseteq \rightarrow_i \subseteq \rightarrow_i$ implies $\rightarrow_i \rightarrow_i \subseteq \rightarrow_i$, And the diamond property of $\rightarrow_i$ implies $\rightarrow_i$ is confluence, thus implies the confluence of $\rightarrow_i$.

Sometimes $\rightarrow$ may not satisfy diamond property, then one can look for the possibility to construct an intermediate reduction $\rightarrow_i$, 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 satisfy diamond property, for example, $((\lambda x.((\lambda y.u) v)) ((\lambda u.y) y) z) \rightarrow_\beta (\lambda x.((\lambda u.u) v)) (z z)$ and $(\lambda x.((\lambda u.u) v) ((\lambda y.y) y) z) \rightarrow_\beta (\lambda u.u) v$. 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).

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

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: $t \Rightarrow_\beta t$ Simply let $t_3$ be $t$.

$\frac{t \Rightarrow_\beta t'}{t' \Rightarrow_\beta t''}$

Case: $\frac{\lambda x.t \Rightarrow_\beta \lambda x.t'}{\lambda x.t \Rightarrow_\beta \lambda x.t'}$

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 $\lambda x.t''$, we get $t_1 \equiv \lambda x.t'' \Rightarrow_\beta \lambda x.t''_3 \equiv t_3$ and $t_2 \equiv \lambda x.t''' \Rightarrow_\beta \lambda x.t'''_3 \equiv t_3$. 

10
By lemma 3, lemma 5 and lemma 6.

Proof. Theorem 3.

If Lemma 7.

diamond property.

contract all the redex in $t$. Based on the notion of parallel reduction, Takahashi [39] observed, instead of trying to prove 5.1.2 Takahashi’s Method

Lemma 6. $\Rightarrow \subseteq \Rightarrow \subseteq \Rightarrow$.

Theorem 3. $\Rightarrow \Rightarrow$ reduction is confluent.


5.1.2 Takahashi’s Method

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

Definition 16. $\Rightarrow \Rightarrow$ is said to satisfy triangle property if: $t \Rightarrow \Rightarrow t’$ implies $t’ \Rightarrow \Rightarrow t^*$, where $t^*$ (we called it parallel contraction) is defined as:

\[
\begin{align*}
x^* & := x. \\
(\lambda x.t)^* & := \lambda x.t^*. \\
(t_1 t_2)^* & := t_1^* t_2^* \text{ if } t_1 t_2 \text{ is not a beta redex.} \\
((\lambda x.t_1) t_2)^* & := [t_2^*/x] t_1^*. 
\end{align*}
\]

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

Lemma 7. If $\Rightarrow \Rightarrow$ has triangle property, then it has diamond property.

Proof.

\[
\begin{array}{c}
t \Rightarrow \Rightarrow t' \Rightarrow \Rightarrow t^*
\end{array}
\]
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: $t \Rightarrow_\beta t$

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.

$$t_1 \Rightarrow_\beta t'_1$$

Case: $\lambda x.t_1 \Rightarrow_\beta \lambda x.t'_1$

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 $\lambda x.t'_1 \Rightarrow_\beta \lambda x.t''_1 \equiv (\lambda x.t_1)^*$.

$$t_4 \Rightarrow_\beta t'_4 \quad t_5 \Rightarrow_\beta t'_5$$

Case: $(\lambda x.t_4) t_5 \Rightarrow_\beta [t'_4/x] t'_5$

t is of the form $(\lambda x.t_4) t_5$, $t'$ is of the form $[t'_4/x] t'_5$, $t_4 \Rightarrow_\beta t'_4$ 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 $[t'_4/x] t'_5 \Rightarrow_\beta [t''_4/x] t''_5 \equiv ((\lambda x.t_4) t_5)^*$ (lemma 4).

$$t_4 \Rightarrow_\beta t'_4 \quad t_5 \Rightarrow_\beta t'_5$$

Case: $t_4 t_5 \Rightarrow_\beta t'_4 t'_5$

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 $t'_4 \Rightarrow_\beta t''_4$ and $t'_5 \Rightarrow_\beta t''_5$. So $t'_4 t'_5 \equiv (\lambda x.t'_6) t'_5 \Rightarrow_\beta [t''_4/x] t''_5 \equiv ((\lambda x.t_6) t_5)^*$.

Assume $t_4$ is not of the form $\lambda x.t_6$. By induction, there is a reduction $t'_4 t'_5 \Rightarrow_\beta t''_4 t''_5 \equiv (t_4 t_5)^*$.

$\square$

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 Lof’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 | \lambda x.t | t t' | (\lambda x.t) t'

We simply label certain beta redexes. Note that $\lambda x.t$ is not a well-formed labelled term, but $(\lambda x.t)t'$ 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).

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

It is natural to make sure that: if $t$ is a well-formed labelled term and $t \Rightarrow_\beta t'$, then $t'$ is also a well-formed labelled term. We can do this by induction on the derivation of $t \Rightarrow_\beta t'$. We will use $\Lambda$ to denote the set of all labelled terms, $\Lambda$ to denote the set of unlabelled terms. So $\Lambda \subset \overline{\Lambda}$. As usual, $\Rightarrow_\beta$ denotes the reflexive and transitive closure of $\Rightarrow_\beta$. $\Rightarrow_\beta$ is defined on terms in $\overline{\Lambda}$. Note that $\Rightarrow_\beta \subseteq \Rightarrow_\beta$.
**Definition 19** (Erasure). We define erasure function \( e : \Lambda \rightarrow \Lambda \) as below:

\[
\begin{align*}
e(x) & := x \\
e(t t') & := e(t) e(t') \\
e(\lambda x. t t') & := \lambda x. e(t t') \\
e((\Delta x. t) t') & := (\lambda x. e(t)) e(t') \\
\end{align*}
\]

graphically denoted by \( \rightarrow_e \)

**Definition 20** (Contraction). We define a contraction function \( \phi : \Lambda \rightarrow \Lambda \) as below:

\[
\begin{align*}
\phi(x) & := x \\
\phi(tt') & := \phi(t) \phi(t') \\
\phi(\lambda x. t t') & := \lambda x. \phi(t t') \\
\phi((\Delta x. t) t') & := [\phi(t')/x] \phi(t) \\
\end{align*}
\]

graphically denoted by \( \rightarrow_\phi \)

The erasure recursively remove all the labels in a term \( t \in \Lambda \) without changing the structure of the term. The contraction functions recursively reduce all the labelled redexes in a labelled term.

**Lemma 9.** If \( t_1 \rightarrow_\beta t_2 \) and \( t'_1 \rightarrow_e t_1 \), then there exist \( t'_2 \) such that \( t'_1 \rightarrow_\beta t'_2 \), and \( t'_2 \rightarrow_e t_2 \).

![Diagram](t_1 \rightarrow_\beta t_2, t'_1 \rightarrow_e t_1, t'_1 \rightarrow_\beta t'_2, t'_2 \rightarrow_e t_2)

**Proof.** If \( t_1 \rightarrow_\beta t_2 \), then \( t_2 \) is obtained by contracting one redex \( \Delta \) in \( t_1 \). We reduce \( \Delta \) (either labelled or unlabelled) in \( t'_1 \) we get \( t'_2 \) (recalled that \( \rightarrow_e \subseteq \rightarrow_\beta \)), which has \( e(t'_2) = t_2 \).

**Lemma 10.**

![Diagram](t_1 \rightarrow_\beta t_2, t'_1 \rightarrow_e t_1, t'_1 \rightarrow_\beta t'_2, t'_2 \rightarrow_e t_2)


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

**Proof.** By induction on the structure of labelled term \( t \).

**Lemma 12.**

![Diagram](t_1 \rightarrow_\beta t_2, \phi(t_1) \rightarrow_\beta \phi(t_2))

**Proof.** By induction on the derivation of \( t_1 \rightarrow_\beta t_2 \). Using lemma 11.
Lemma 13.

\[ t \mapsto e(t) \mapsto \beta \mapsto \phi(t) \]

Proof. By induction on the structure of \( t \).

Lemma 14 (Strip Lemma).

\[ t \mapsto t_1 \mapsto t_2 \mapsto t' \]

Proof. Let \( t_1 \) be the result of reducing the redex \( \Delta \) in \( t \). Let \( t_3 \in A \) be the term obtained from \( t \) by indexing \( \Delta \). So \( \phi(t_3) \equiv t_1 \). By the following diagram:

\[ t \mapsto t_1 \mapsto t_3 \mapsto t_2 \mapsto t' \]

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 \( t \rightarrow_\beta t_1 \), there exist \( t_3 \in A \) 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 \( \lambda x.t \ x \rightarrow_\eta t \). 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 \( \phi \) and \( e \) unchanged, consider \( \lambda x. (\lambda y.y \ z) \ x \rightarrow_\eta \lambda y.y \ z \). Since it does not construct a beta redex, if we let \( t_3 \equiv \lambda x. (\lambda y.y \ z) \ x \in A \), 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, \( (\Lambda, \{\rightarrow_\beta, \rightarrow_\eta\}) \). Confluence problem for this kind of system require some nontrivial efforts to prove. Hardin’s interpretation method [25] provide a way to deal with some of those reduction systems.
Lemma 15 (Interpretation lemma). Let → be →₁ ∪ →₂, →₁ being confluent and strongly normalizing. We denote by ν(a) the →₁-normal form of a. Suppose that there is some relation →ᵣ on →₁ normal forms satisfying:

→ᵣ ⊆ →₁ and a →₂ b implies ν(a) →ᵣ ν(b) (†)

Then the confluence of →ᵣ implies the confluence of →.

Proof. So suppose →ᵣ is confluent. If a →₁ a' and a →₁ a''. So by (†), ν(a) →ᵣ ν(a') and ν(a) →ᵣ ν(a''). Notice that t →ᵣ t' implies ν(t) = ν(t')(By confluence and strong normalizing of →₁). By confluence of →ᵣ, there exists b such that ν(a') →ᵣ b and ν(a'') →ᵣ b. Since →ᵣ ⊆ →₁, we get a' →ᵣ ν(a') →ᵣ b and a'' →ᵣ ν(a'') →ᵣ b. Hence → is confluent.

\[\begin{array}{c}
\vdots \\
\quad a
\end{array}\]

\[\begin{array}{cccc}
a & \rightarrow & \nu(a) & \rightarrow \nu(a')
\end{array}\]

\[\begin{array}{cccc}
\downarrow \\
b
\downarrow \\
\nu(a'')
\end{array}\]

Hardin’s method reduce the confluence problem of →₁ ∪ →₂ to →ᵣ, given the confluence and strong normalizing of →₁, this make it possible to apply Tait-Martin-Löf’s (Takahashi’s) method to prove confluence of →ᵣ.

5.2.1 Local λμ Calculus

We now show an application 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 | λx.t | tt' | μt
Closure μ ::= \{xᵢ → tᵢ\}ᵢ∈I

The closure is basically a set of recursively defined definitions. Let I be a finite nonempty index set. For \{xᵢ → tᵢ\}ᵢ∈I, we require for any 1 ≤ i ≤ n, the set of free variables of tᵢ, FV(tᵢ) ⊆ dom(μ) = \{x₁,...,xₙ\} 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). μ ∈ t means the closure μ appears in t. jμt denotes μ₁...μₙ.t. \[(t'/x)(μt) \equiv μ(\{t'/x\}t)\]. So FV(μt) = FV(t) - dom(μ).

Definition 22 (Beta-Reductions).
\[
\begin{array}{ccc}
(\lambda x.t)t' \rightarrow _β [t'/x] & \quad (xᵢ \rightarrow tᵢ) \in μ & \quad \mu xᵢ \rightarrow _β μtᵢ \\
λx.t \rightarrow _β λx.t' & \quad t \rightarrow _β t' & \quad t \rightarrow _β t'' \\
λx.t \rightarrow _β t'' & \quad t \rightarrow _β t' & \quad t \rightarrow _β t'
\end{array}
\]

15
Lemma 17. $\Rightarrow_{\mu}$ is strongly normalizing and confluent.

Definition 24 ($\mu$-Normal Forms).

Let $x, \mu x_i : \lambda x. n \mid mn'$.

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

Definition 25 ($\mu$-Normalize Function).

$\frac{m(x) := x \quad m(\lambda y. t) := \lambda y. m(t) \quad m((\mu t_1)t_2) := m(t_1)m(t_2) \quad m(\mu t_1) := \mu y \text{ if } y \notin \text{dom}(\mu) \quad m(\mu t_1(t_2)) := m(\mu t_1)m(\mu t_2) \quad m(\mu(\lambda x.t)) := \lambda x.m(\mu t)}{m(t)\Rightarrow_{\mu} m(t)}$

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

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

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 $n_1 \Rightarrow_{\mu} n_1'$ and $n_2 \Rightarrow_{\mu} n_2'$, then $m((n_2/x)n_1) \Rightarrow_{\mu} m((n_2'/x)n_1')$.

Lemma 22. If $n \Rightarrow_{\mu} n'$ and $n \Rightarrow_{\mu} n''$, then there exist $n'''$ such that $n'' \Rightarrow_{\mu} n'''$ and $n' \Rightarrow_{\mu} n'''$. So $\Rightarrow_{\mu}$ is confluent.

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

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

Theorem 4. $\Rightarrow_{\beta} \cup \Rightarrow_{\mu}$ 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 \( t \rightarrow t' \), then \( \Gamma \vdash t' : T \). In dependent type system, the following conversion rule is presented:

\[
\frac{\Gamma \vdash t : T' \quad T = T'}{\Gamma \vdash t : T} \quad \text{Conv}
\]

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 \( \Gamma \vdash \lambda x. t_1 : T_1 \rightarrow T_2 \), \( \Gamma \vdash t_2 : T_1 \) and \( T = T' \). Since \((\lambda x. t_1) t_2 \rightarrow [t_2/x]t_1\), we need to show \( \Gamma \vdash [t_2/x]t_1 : T \). \( \Gamma \vdash \lambda x. t_1 : T_1 \rightarrow T_2 \) implies \( \Gamma, x : T_1 \vdash t_1 : T_2' \) and \( T_1' \rightarrow T_2' = T_1 \rightarrow T_2 \). 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 \( T_1' \rightarrow T_2' = T_1 \rightarrow T_2 \), is it true that \( T_1' = T_1 \) and \( T_2' = T_2' \)? We called this inverse structure congruence problem. We know that given \( T_1' = T_1 \) and \( T_2' = T_2 \), one can conclude that \( T_1' \rightarrow T_2' = T_1 \rightarrow T_2 \). It is not immediate that the inverse structure congruence holds, so we need to analyze the convertability relation between \( T_1' \rightarrow T_2' \) and \( T_1 \rightarrow T_2 \). We would like the following invertability property holds.

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

A reducton system \((T, \rightarrow)\), where \( T \) is a set of types, arise when we analyze the relation \( T = T' \). Often for such system we want to design \( \rightarrow \) such that \( T_1 \rightarrow T_2 \) can only be reduced to \( T_1' \rightarrow T_2' \) when \( T_1 \rightarrow T_1' \) or \( T_2 \rightarrow T_2' \). So confluence of \((T, \rightarrow)\) will imply that, for \( T_1' \rightarrow T_2' = T_1 \rightarrow T_2 \), there is a \( T_3 \) such that \( T_1 \rightarrow T_2 \rightarrow T_3 \) and \( T_1' \rightarrow T_2' \rightarrow T_3' \). So we know \( T_3 \) must be of the form \( T_4 \rightarrow T_5 \). So \( T_1 \rightarrow T_4 \rightarrow T_1' \) and \( T_2 \rightarrow T_5 \rightarrow T_2' \), 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 Selfstar, but we will have to leave that to future work.

6 Future Explorations and Conclusions

6.1 System 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 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 \( \mu ::= \{ x_i \mapsto t_i \}_{i \in \mathbb{I}} \).

Context \( \Gamma ::= \emptyset \mid \Gamma, x : t \mid \Gamma, \check{\mu} \).

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

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).
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 recursive definition.

**Definition 31** (Church Encoding). Let \( \bar{\mu}_c \) be the following recursive definitions:

\[
\begin{align*}
\bar{\mu}_c, C : \text{Nat} & \to *, s : (\Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n))) \to (C \ 0) \to (C \ x) \\
\bar{\mu}_c, C : \text{Nat} & \to *, s : (\Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n))), z : C \ 0 \to z : C \ (\lambda C.\lambda s.\lambda z.\, z) \\
\bar{\mu}_c, C : \text{Nat} , \lambda C.\lambda s.\lambda z.\, : \text{Nat} & \to *, s : (\Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n))) \to (C \ 0) \to (C \ (\lambda C.\lambda s.\lambda z.\,)) \\
\bar{\mu}_c, n : \text{Nat} & \to \lambda n.\lambda C.\lambda s.\lambda z.\, n \ (n \ C \ s \ z) \\
(0 : \text{Nat}) & \to \lambda C.\lambda s.\lambda z.\, \\
\end{align*}
\]

Now let us see how we can derive \( \bar{\mu}_c \vdash \lambda C.\lambda s.\lambda z.\, : \text{Nat} \) and \( \bar{\mu}_c \vdash \lambda n.\lambda C.\lambda s.\lambda z.\, n \ (n \ C \ s \ z) : \text{Nat} \to \text{Nat} \).

\[
\begin{align*}
\Delta_1 & \\
\bar{\mu}_c, n : \text{Nat}, \Gamma & \vdash s \ n : (C \ n) \to (C \ (S \ n)) \quad \Delta_2 \\
\bar{\mu}_c, n : \text{Nat}, C : \text{Nat} & \to *, s : (\Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n))), z : C \ 0 \to s \ n : (n \ C \ s \ z) : C \ (S \ n) \\
\bar{\mu}_c, n : \text{Nat} & \to \lambda C.\lambda s.\lambda z.\, n \ (n \ C \ s \ z) : \Pi C : \text{Nat} \to *, s : (\Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n))) \to (C \ 0) \to (C \ (\lambda C.\lambda s.\lambda z.\,)) \\
\bar{\mu}_c, n : \text{Nat} & \vdash \lambda n.\lambda C.\lambda s.\lambda z.\, n \ (n \ C \ s \ z) \\
\end{align*}
\]

In above derivation, \( \Gamma = C : \text{Nat} \to *, s : (\Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n))), z : C \ 0 \), and \( \iota \) step first convert \( S \ n \) to \( \lambda C.\lambda s.\lambda z.\, n \ (n \ C \ s \ z) \), then apply the SelfGen rule. The \( \Delta_1 \) is a subderivation:

\[
\begin{align*}
\bar{\mu}_c, n : \text{Nat}, \Gamma & \vdash s \ : \Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n)) \quad \Delta_1 \\
\bar{\mu}_c, n : \text{Nat}, \Gamma & \vdash n : \text{Nat} \\
\end{align*}
\]

The \( \Delta_2 \) is a subderivation:

\[
\begin{align*}
\bar{\mu}_c, n : \text{Nat}, \Gamma & \vdash n : \text{Nat} \\
\bar{\mu}_c, n : \text{Nat}, \Gamma & \vdash \iota : \Pi C : \text{Nat} \to *, s : (\Pi n : \text{Nat}.(C \ n) \to (C \ (S \ n))) \to (C \ 0) \to (C \ x) \\
\bar{\mu}_c, n : \text{Nat}, \Gamma & \vdash \iota : \text{Nat} \\
\end{align*}
\]

18
where $\Delta_3$:

\[
\mu_{c,n} : \text{Nat} \vdash C : \text{Nat} \rightarrow \text{Var} \quad \mu_{c,n} : \text{Nat} \vdash s : \Pi n : \text{Nat}.(C\ n) \rightarrow (C\ (S\ n)) \quad \text{Var} \quad \mu_{c,n} : \text{Nat} \vdash z : C\ 0 \quad \text{Var}
\]

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:

\[
\mu_c \vdash (\text{IIC} : \text{Nat} \rightarrow *, \Pi n : \text{Nat}.((C\ n) \rightarrow (C\ (S\ n)))) \rightarrow C\ 0 \rightarrow \Pi n : \text{Nat}.C\ n.
\]

With $*:*$, we now can define Leibniz's equality as $\text{Eq} := \lambda A.\lambda x.\lambda y.\text{IIC} : (A \rightarrow *).C\ x \rightarrow C\ y$. Now we have the judgement $\mu_c \vdash \text{Eq} : \Pi A : *, A \rightarrow A \rightarrow *$. Now define addition:

\[
\mu_c \vdash (\text{add} := \lambda n.\lambda m.\text{Ind}\ (\lambda y.\text{Nat})\ (\lambda x.\text{S})\ m\ n) : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}.
\]

Now one can use induction principle to derive $\mu_c \vdash t : \Pi m : \text{Nat}.(\text{Eq\ Nat\ (add\ m\ 0)\ m})$ for some term $t$, the term $t$ is a lambda expression that encode a proof of the formula $\Pi m : \text{Nat}.(\text{Eq\ Nat\ (add\ m\ 0)\ m})$ (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 $\mu_c \vdash t : \Pi m : \text{Nat}.(\text{Eq\ Nat\ (add\ m\ 0)\ m})$, 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 $\mu_s$ be the following recursive definitions:

\[
\begin{align*}
& (\text{Nat} : *) \mapsto \lambda x.\Pi C : \text{Nat} \rightarrow * : \Pi (\Pi n : \text{Nat}.C\ (S\ n)) \rightarrow (C\ 0) \rightarrow (C\ x) \\
& (S : \text{Nat} \rightarrow \text{Nat}) \mapsto \lambda n.\lambda C.\lambda s.\lambda z.\text{r} n
\\
& (0 : \text{Nat}) \mapsto \lambda C.\lambda s.\lambda z. z
\end{align*}
\]

With Scott numerals defined above, one can derive a case analysis principle:

\[
\mu_s \vdash \text{Case} := \lambda C.\lambda s.\lambda z.\lambda n.\text{r} n : C\ s\ z) : \Pi C : \text{Nat} \rightarrow * : \Pi (\Pi n : \text{Nat}.(C\ (S\ n)) \rightarrow (C\ 0) \rightarrow \Pi n : \text{Nat}.C\ n
\]

addition function can also be defined by extending the closure $\mu_s$ by

\[
\begin{align*}
& \text{add} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
& \text{add} \vdash \text{add} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
& \text{add} \vdash \lambda n.\lambda m.\text{Case} (\lambda n.\text{Nat}) (\lambda p.\text{S}\ (\text{add}\ p\ m)) m \ n
\end{align*}
\]

One can further prove theorems about the $\text{add}$ function like what we did for Church encoding, we will not pursue that here. Interestingly the expression for $\text{Case}$ and $\text{Ind}$ are the same, they are used in the $\text{add}$ 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 $\lambda$ 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


A Proofs

A.1 Proof of Lemma 17

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

Proof. One way to prove this is first identify $t$ as $\mu t'$, here $\mu t'$ 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 $m(\mu t') \equiv \lambda x.m(t'')$. Now we can again identify $t''$ as $\mu t''$, where $t''$ does not have any closure at head position. Since $t''$ is structurally smaller than $\lambda x.t''$, by IH, $m(\mu t'') \in \Phi$, thus $m(\mu t') \equiv \lambda x.m(t'') \in \Phi$.

For $t' = t_1t_2$, 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 $m(\lambda y.[n_2/x]n) \equiv \lambda y.m([n_2/x]n) \Rightarrow_{\beta\mu} \lambda y.m([n'_2/x]n) \equiv m(\lambda y.[n'_2/x]n)$.

Step Case: $n_1 = mn'$. We have $m([n_2/x]n_2/x)n') \equiv m([n_2/x]n)m([n_2/x]) \Rightarrow_{\beta\mu} m([n'_2/x]n)m([n'_2/x]) \Rightarrow m([n'_2/x]n[nn'/x])$. 

\qed
A.3 Proof of Lemma 20

\[ m(m(t)) \equiv m(t) \text{ and } m([m(t_1)/y]m(t_2)) \equiv m([t_1/y]t_2). \]

**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 \( \mu \bar{t}_2 \), \( \bar{t}_2 \) does not contains any closure at head position. We proceed by induction on the structure of \( \bar{t}_2 \):

**Base Cases:** For \( \bar{t}_2 = x \), we use \( m(m(t)) \equiv m(t) \).

**Step Cases:** If \( \bar{t}_2 = \lambda x.t''_{2} \), then \( m(\mu \bar{t}_2 (\lambda x.[t_1/y]t''_{2})) \equiv \lambda x.m(\mu \bar{t}_2 (\lambda x.[t_1/y]t''_{2})) \equiv \lambda x.m(\mu \bar{t}_2 (\lambda x.[t_1/y]t''_{2})) \), where \( t''_{2} \) as \( \mu \bar{t}_2 t''_{2} \) and \( t''_{2} \) does not have any closure at head position. Since \( t''_{2} \) is structurally smaller than \( \lambda x.t''_{2} \), by IH, \( m(\mu \bar{t}_2 (\lambda x.[t_1/y]t''_{2})) \equiv m([t_1/y]m(\mu \bar{t}_2 t''_{2})) \). Thus \( \lambda x.m(\mu \bar{t}_2 (\lambda x.[t_1/y]t''_{2})) \equiv \lambda x.m([t_1/y]m(\mu \bar{t}_2 t''_{2})) \). So \( m([t_1/y]m(\mu \bar{t}_2 (\lambda x.t''_{2}))) \equiv m([m(t_1)/y]m(\lambda x.m(\mu \bar{t}_2 t''_{2}))) \equiv m([m(t_1)/y]m(\lambda x.m(\mu \bar{t}_2 t''_{2}))) \).

For \( \bar{t}_2 = t_3 t_4 \), we can argue similarly as above. \( \square \)

A.4 Proof of Lemma 21

If \( n_1 \Rightarrow_{\beta} n'_1 \) and \( n_2 \Rightarrow_{\beta} n'_2 \), then \( m([n_2/y]n_1) \Rightarrow_{\beta} m([n'_2/y]n'_1) \).

**Proof.** We prove this by induction on the derivation of \( n_1 \Rightarrow_{\beta} n'_1 \).

**Base Case:**

\( n \Rightarrow_{\beta} n \)

By the lemma 19.

**Base Case:**

\[
x_i \mapsto t_i \in \mu \\
\mu x_i \Rightarrow_{\beta} m(\mu x_i)
\]

Because \( y \notin \text{FV}(\mu x_i) \) and \( \mu \) is local.

**Step Case:**

\[
n_a \Rightarrow_{\beta} n'_a \quad n_b \Rightarrow_{\beta} n'_b
\]

\[
(\lambda x.n_a)n_b \Rightarrow_{\beta} m([n'_a/x]n'_b)
\]

We have \( m(\lambda x.[n_2/y]n_1) \equiv (\lambda x.m([n_2/y]n_1))m([n_2/y]n_b) \)

\[
\Rightarrow_{IH} m([m([n'_2/y]n'_b/x]m([n'_2/y]n'_a))) \equiv m([n'_2/y]([n'_b/x]n'_a))). \text{ The last equality is by lemma 20.}
\]

**Step Case:**

\[
n \Rightarrow_{\beta} n'
\]

\[
\lambda x.n \Rightarrow_{\beta} \lambda x.n'
\]

We have \( m(\lambda x.[n_2/y]n) \equiv \lambda x.m([n_2/y]n) \Rightarrow_{IH} \lambda x.m([n'_2/y]n')) \equiv m(\lambda x.[n'_2/y]n') \)

23
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 \(m([n_2/y]n_a[n_2/y]n_b) \equiv m([n_2/y]n_a)m([n_2/y]n_b)\)
\[
\Rightarrow_{\beta \mu} m([n'_2/y]n'_a)m([n'_2/y]n'_b) \equiv m([n'_2/y](n'_a n'_b)).
\]

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:

\(n \Rightarrow_{\beta \mu} n\)

Obvious.

Base Case:

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

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 \(n''_2 \Rightarrow_{\beta \mu} n''_2\). 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. \(m(\bar{\mu} \bar{t}) \equiv m(\bar{\mu} \bar{t})\) and \(m(\bar{\mu}([t_2/x]t_1)) \equiv m([\bar{\mu}t_2/x] \bar{t}t_1)\)

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

Proposition 1. If \(a \rightarrow_{\beta} b\), then \(m(a) \rightarrow_{\beta} m(b)\).

Proof. We prove this by induction on the derivation(depth) of \(a \rightarrow_{\beta} b\). We list a few non-trivial cases:

Base Case:
We have $m(\mu x_i) \equiv \mu x_i \rightarrow_{\beta} m(\mu t_i)$.

**Base Case:**

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

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

**Step Case:**

$$t \rightarrow_{\beta} t'$$

$$\lambda x.t \rightarrow_{\beta} \lambda x.t'$$

By IH, we have $m(\lambda x.t) \equiv \lambda x.m(t) \rightarrow_{\beta} \lambda x.m(t') \equiv m(\lambda x.t')$.

**Step Case:**

$$t \rightarrow_{\beta} t'$$

$$\mu t \rightarrow_{\beta} \mu t'$$

We want to show $m(\mu t) \rightarrow_{\beta} m(\mu t')$. If $dom(\mu) \not\subseteq FV(t)$, then $m(\mu t) \equiv m(t) \rightarrow_{\beta} m(t') \equiv m(\mu t')$. 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 $\mu t''$, where $t''$ does not contain any closure at head position. We do case analyze on the structure of $t''$:

**Case.** $t'' = x_i \in dom(\mu t)$ or $x_i \notin dom(\mu t)$, these cases will not arise.

**Case.** $t'' = \lambda y.t_1$, then it must be that $t' = \mu t_2(\lambda y.t_1)$ where $t_1 \rightarrow_{\beta} t'_1$. So we get $\mu t_2 \rightarrow_{\beta} \mu t_2'$.

By IH(depth of $\mu t_2 \rightarrow_{\beta} \mu t_2'$ is smaller), we have $m(\mu t_2) \rightarrow_{\beta} m(\mu t_2')$. Thus $m(\mu t_2(\lambda y.t_1)) \equiv \lambda y.m(\mu t_2(\lambda y.t_1)) \equiv m(\mu t_2(\lambda y.t_1)).$

**Case.** $t'' = t_1t_2$ and $t' = \mu t_2(t_1't_2)$, where $t_1 \rightarrow_{\beta} t'$. We have $\mu t_2 \rightarrow_{\beta} \mu t_2'$. By IH(depth of $\mu t_2 \rightarrow_{\beta} \mu t_2'$ is smaller), $m(\mu t_2) \rightarrow_{\beta} m(\mu t_2')$.

Thus $m(\mu t_2(t_1t_2)) \equiv m(\mu t_2)m(\mu t_2(t_1t_2)) \rightarrow_{\beta} m(\mu t_2(m(\mu t_2(t_2))). For t'' = t_1t_2, where $t_2 \rightarrow_{\beta} t_2'$, we can argue similarly.

**Case.** $t'' = (\lambda y.t_1)t_2$ and $t' = \mu t_2(\lambda y.t_1)$.

Thus $m(\mu t_2((\lambda y.t_1)t_2)) \equiv (\lambda y.m(\mu t_2(t_1)))m(\mu t_2(t_2)) \rightarrow_{\beta} m([m(\mu t_2)/y]m(\mu t_2(t_1))) \equiv m([m(\mu t_2)/y][m(\mu t_2)]) \equiv m(\mu t_2(t_1))(\text{lemma 24}).$