Self Types for Dependently Typed Lambda Encodings

Peng Fu, Aaron Stump

Computer Science, The University of Iowa

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

1 Introduction

Modern type-theoretic tools Coq and Agda extend a typed lambda calculus with a rich notion of primitive datatypes. Both tools build on established foundational concepts, but the interactions of these, particularly with datatypes and recursion, often leads to unexpected problems. For example, it is well-known that type preservation does not hold in Coq, due to the treatment of coinductive types [14]. Arbitrary nesting of coinductive and inductive types is not supported by the current version of Agda, leading to new proposals like co-patterns [2]. And new issues are discovered with disturbing frequency; e.g., an unexpected incompatibility of extensional consequences of Homotopy Type Theory with both Coq and Agda was discovered in December, 2013 [20].

The above issues all are related to the datatype system, which must determine what are the legal inductive/coinductive datatypes, in the presence of indexing, dependency, and generalized induction (allowing functional arguments to constructors). And for formal study of the type theory – either on paper [22], or in a proof assistant [5] – one must formalize the datatype system, which can be daunting, even in very capable hands (cf. Section 2 of [6]).

Fortunately, an alternative to primitive datatypes exists: lambda encodings, like the well-known Church and Scott encodings [7,10]. Utilizing the core typed lambda calculus for representing data means that no datatype system is needed at all, greatly simplifying the formal theory. We focus here just on inductive types, since in extensions of System **F**, coinductive types can be reduced to inductive ones [12].

Several problems historically prevented lambda encodings from being adopted in practical type theories. Scott encodings are efficient but do not inherently provide a form of iteration or recursion. Church encodings inherently provide iteration, and are typable in System **F**. Due to strong normalization of System **F** [15], they are thus suitable for use in a total (impredicative) type theory, but:

1. The predecessor of n takes $O(n)$ time to compute instead of constant time.
2. We cannot prove $0 \neq 1$ with the usual definition of $\neq$.
3. Induction is not derivable [13].
4. Large eliminations (computing types from data) are not supported.

These issues motivated the development of the Calculus of Inductive Constructions (cf. [21]). Problem (1) is best known but has a surprisingly underappreciated solution: if we accept positive recursive definitions (which preserve normalization), then we can use Parigot numerals, which are like Church numerals but based on recursors not iterators [19]. Normal forms of Parigot numerals are exponential in size, but a reasonable term-graph implementation should be able to keep them linear via sharing. The other three problems have remained unsolved.

In this paper, we propose solutions to problems (2) and (3). For problem (2) we propose to change the definition of falsehood from explosion ($\forall X.X$, everything is true) to equational inconsistency ($\forall X.\Pi x : X.\Pi y : X.x =_X y$, everything is equal for any type). We point out that $0 \neq 1$ is derivable with this notion. Our main contribution is for problem (3). We adapt **CC** to support dependent elimination with Church or Parigot encodings, using a novel type construct called *self types*, $\iota x.T$, to express dependency of a type on its subject. This allows deriving induction principles in a total type theory, and we believe it is the missing piece of the puzzle for dependent typing of pure lambda calculus.

We summarize the main technical points of this paper:

- System **S**, which enables us to encode Church and Parigot data and derive induction principles for these data.
- We prove strong normalization of **S** by erasure to a version of $\mathbf{F}_\omega$ with positive recursive type definitions. We prove strong normalization of this version of $\mathbf{F}_\omega$ by adapting a standard argument.
- Type preservation for **S** is proved by extending Barendregt's method [4] to handle implicit products and making use of a confluence argument.

Detailed arguments omitted here may be found in an extended version [11].

2 Overview of System S

System **S** extends a type-assignment formulation of the Calculus of Constructions (**CC**) [9]. We allow global recursive definitions in a form we call a *closure*:

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

The x_i are term variables which cannot appear in the terms t_i, but can appear in the types T_i. Occurrences in types are used to express dependency, and are

crucial for our approach. Erasure to $\mathbf{F}_\omega$ with positive recursive definitions will drop all such occurrences. The X_i are type variables that can appear positively in the T_i or at erased positions (explained later).

The essential new construct is the self type $\iota x.T$. Note that this is different from self typing in the object-oriented (OO) literature, where the central problem has been to allow self-application while still validating natural record-subtyping rules [18,1]. Typing the self parameter of an object's methods appears different from allowing a type to refer to its subject, though Hickey proposes a type-theoretic encoding of objects based on very dependent function types $\{f \mid x : A \to B\}$, where the range B can depend on both x and values of the function f itself [16]. The self types we propose appear to be simpler.

2.1 Induction Principle

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

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

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

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

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

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

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

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

These equalities suggest that $\mathsf{Ind} := \lambda x.\lambda f.\lambda a.x\ f\ a$, using Parigot numerals [19]:

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

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

Each numeral corresponds to its terminating recursor.

Now, let us try to type these lambda numerals. It is reasonable to assign $s : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y))$ and $z : P\ \bar{0}$. Thus we have the following typing relations:

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

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

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

So we want to define Nat to be something like:

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

Two problems arise with this scheme of encoding. The first problem involves recursiveness. The definiens of Nat contains Nat and $\mathsf{S}, \bar{0}$, while the type of S is $\mathsf{Nat} \to \mathsf{Nat}$ and the type of $\bar{0}$ is Nat. So the typing of Nat will be mutually

recursive. Observe that the recursive occurrences of Nat are all at the type-annotated positions; i.e., the right side of the ":".

Note that the subdata of $\bar{n}$ is responsible for one recursive occurrence of Nat, namely, $\Pi y : \mathsf{Nat}$. If one never computes with the subdata, then these numerals will behave just like Church numerals. This inspires us to use Miquel's implicit product [17]. In this case, we want to redefine Nat to be something like:

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

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

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

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

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

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

We have the following inferences[1]:

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

2.2 The Notion of Contradiction

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

On the other hand, an equational theory is considered inconsistent if $a = b$ for all term a and b. So we propose to use $\forall A : *.\Pi x : A.\Pi y : A.x =_A y$ as

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

the notion of contradiction in **CC**. We first want to make sure it is uninhabited. The way to argue that is first assume it is inhabited by t. Since **CC** is strongly normalizing, the normal form of t must be of the form[2] $[\lambda A : *.]\lambda x[: A].\lambda y[: A].[\lambda C : A \to *].\lambda z[: C\ x].n$ for some normal term n with type $C\ y$, but we know that there is no combination of x, y, z to make a term of type $C\ y$. So the type $\forall A : *.\Pi x : A.\Pi y : A.\forall C : A \to *.Cx \to Cy$ is uninhabited. We can then prove the following theorem [3]:

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

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

3 System S

We use gray boxes to highlight the terms, types and rules that are not in $\mathbf{F}_\omega$ with positive recusive definitions [4].

3.1 Syntax

$$
\begin{array}{lll}
\textit{Terms } t & ::= & x \mid \lambda x.t \mid tt' \\
\textit{Types } T & ::= & X \mid \forall X : \kappa.T \mid \Pi x : T_1.T_2 \mid \boxed{\forall x : T_1.T_2} \mid \\
 & & \boxed{\iota x.T} \mid \boxed{T\ t} \mid \lambda X.T \mid \boxed{\lambda x.T} \mid T_1 T_2 \\
\textit{Kinds } \kappa & ::= & * \mid \boxed{\Pi x : T.\kappa} \mid \Pi X : \kappa'.\kappa \\
\textit{Context } \Gamma & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, X : \kappa \mid \Gamma, \mu \\
\textit{Closure } \mu & ::= & \{(x_i : S_i) \mapsto t_i\}_{i \in N} \cup \{(X_i : \kappa_i) \mapsto T_i\}_{i \in M}
\end{array}
$$

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

Legal positions for recursion in closures. For $\{(x_i : S_i) \mapsto t_i\}_{i \in N}$, we do not allow any recursive (or mutually recursive) definitions. For $\{(X_i : \kappa_i) \mapsto T_i\}_{i \in M}$, we only allow singly recursive type definitions, but not mutually recursive ones. This is not a fundamental limitation of the approach; it is just for simplicity of the normalization argument. The recursive occurrences of type variables can only be at positive or erased positions. Erased positions, following the erasure function we will see in Section 5.1, are those in kinds or in the types for $\forall$-bound variables.

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

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

[3] Coq code for this is in the extended version.

[4] Full specification of $\mathbf{F}_\omega$ with positive recursive definitions is in the extended version.

3.2 Kinding and Typing

Some remarks on the typing and kinding rules:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Lambda Encodings in S

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

4.1 Natural Numbers

Definition 1 (Church Numerals). *Let μ_c be the following closure:*
$(\mathsf{Nat} : *) \mapsto \iota x.\forall C : \mathsf{Nat} \to *.(\forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n)) \to C\ 0 \to C\ x$
$(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$
$(0 : \mathsf{Nat}) \mapsto \lambda s.\lambda z.z$

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

Definition 2 (Parigot Numerals). *Let μ_p be the following closure:*
$(\mathsf{Nat} : *) \mapsto \iota x.\forall C : \mathsf{Nat} \to *.(\ \Pi\ n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n)) \to C\ 0 \to C\ x$
$(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda s.\lambda z.s\ \ n\ \ (n\ s\ z)$
$(0 : \mathsf{Nat}) \mapsto \lambda s.\lambda z.z$

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

Theorem 2 (Induction Principle).
$\mu_c \vdash \mathsf{Ind} : \forall C : \mathsf{Nat} \to *.(\forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n)) \to C\ 0 \to \Pi n : \mathsf{Nat}.C\ n$
where $\mathsf{Ind}\ := \lambda s.\lambda z.\lambda n.n\ s\ z$
with $s : \forall n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n), z : C\ 0, n : \mathsf{Nat}.$

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

It is worth noting that it is really the definition of Nat and the *selfInst* rule that give us the induction principle, which is not derivable in **CC** [8].

Definition 3 (Addition). $m + n := \mathsf{Ind}\ \mathsf{S}\ n\ m$

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

Definition 4 (Leibniz's Equality). $\mathsf{Eq} := \lambda A[: *].\lambda x[: A].\lambda y[: A].\forall C : A \to *.C\ x \to C\ y.$

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

Theorem 3. $\mu_c \vdash \Pi x : \mathsf{Nat}.x + 0 =_{\mathsf{Nat}} x$

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

The above theorem is provable inside **S**. It shows how to inhabit the type $\Pi x : \mathsf{Nat}.x + 0 =_{\mathsf{Nat}} x$ given μ_c, using Ind.

4.2 Vector Encoding

Definition 5 (Vector). *Let μ_v be the following definitions:*
$(\mathsf{vec} : * \to \mathsf{Nat} \to *) \mapsto$
$\lambda U : *.\lambda n : \mathsf{Nat}.\ \iota x\ .\forall C :\ \Pi p : \mathsf{Nat}.\mathsf{vec}\ U\ p \to *\ .$
$(\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(C\ m\ y\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ y)))$
$\to C\ 0\ \mathsf{nil} \to C\ n\ \ x$
$(\mathsf{nil} : \forall U : *.\mathsf{vec}\ U\ 0) \mapsto \lambda y.\lambda x.x$
$(\mathsf{cons} : \Pi n : \mathsf{Nat}.\forall U : *.U \to \mathsf{vec}\ U\ n \to \mathsf{vec}\ U\ (\mathsf{S}\ n)) \mapsto \lambda n.\lambda v.\lambda l.\lambda y.\lambda x.y\ n\ v\ (l\ y\ x)$
where $n : \mathsf{Nat}, v : U, l : \mathsf{vec}\ U\ n, y : \Pi m : \mathsf{Nat}.\Pi u : U.\forall z : \mathsf{vec}\ U\ m.(C\ m\ z\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ z)), x : C\ 0\ \mathsf{nil}.$

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

Definition 6 (Induction Principle for Vector).
$\mu_v \vdash \mathsf{Ind} :$
$\forall U : *.\Pi n : \mathsf{Nat}.\forall C : \mathsf{Nat} \to \mathsf{vec}\ U\ p \to *.$
$(\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(C\ m\ y\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ y)))$
$\to C\ 0\ \mathsf{nil} \to \Pi x : \mathsf{vec}\ U\ n.(C\ n\ x)$
where $\mathsf{Ind} := \lambda n.\lambda s.\lambda z.\lambda x.x\ s\ z$
$n : \mathsf{Nat}, s : \forall C : (\mathsf{Nat} \to \mathsf{vec}\ U\ p \to *).(\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(C\ m\ y\ \to C\ (\mathsf{S}\ m)\ (\mathsf{cons}\ m\ u\ y))), z : C\ 0\ \mathsf{nil}, x : \mathsf{vec}\ U\ n.$

Definition 7 (Append).
$\mu_v \vdash \mathsf{app} : \forall U : *.\Pi n_1 : \mathsf{Nat}.\Pi n_2 : \mathsf{Nat}.\mathsf{vec}\ U\ n_1 \to \mathsf{vec}\ U\ n_2 \to \mathsf{vec}\ U\ (n_1 + n_2)$
where $\mathsf{app} := \lambda n_1.\lambda n_2.\lambda l_1.\lambda l_2.(\mathsf{Ind}\ n_1)\ (\lambda n.\lambda x.\lambda v.\mathsf{cons}\ (n + n_2)\ x\ v)\ l_2\ l_1$.

Typing: We want to show $\mathsf{app} : \forall U : *.\Pi n_1 : \mathsf{Nat}.\Pi n_2 : \mathsf{Nat}.\mathsf{vec}\ U\ n_1 \to \mathsf{vec}\ U\ n_2 \to \mathsf{vec}\ U\ (n_1 + n_2)$. Observe that $\lambda n.\lambda x.\lambda v.\mathsf{cons}(n + n_2)\ x\ v : \Pi n : \mathsf{Nat}.\Pi x : U.\mathsf{vec}\ U\ (n + n_2) \to \mathsf{vec}\ U\ (n + n_2 + 1)$. We instantiate $C := \lambda y.(\lambda x.\mathsf{vec}\ U\ (y + n_2))$, where x free over $\mathsf{vec}\ U\ (y + n_2)$, in $\mathsf{Ind}\ n_1$. By beta reductions, we get $\mathsf{Ind}\ n_1 : (\Pi m : \mathsf{Nat}.\Pi u : U.\forall y : \mathsf{vec}\ U\ m.(\mathsf{vec}\ U\ (m + n_2) \to \mathsf{vec}\ U\ ((\mathsf{S}\ m) + n_2)) \to \mathsf{vec}\ U\ (0 + n_2) \to \Pi x : \mathsf{vec}\ U\ n_1.\mathsf{vec}\ U\ (n_1 + n_2)$.
So $(\mathsf{Ind}\ n_1)\ (\lambda n.\lambda x.\lambda v.\mathsf{cons}(n+n_2)\ x\ v) : \mathsf{vec}\ U\ (0+n_2) \to \Pi x : \mathsf{vec}\ U\ n_1.\mathsf{vec}\ U\ (n_1+n_2)$.
We assume $l_1 : \mathsf{vec}\ U\ n_1, l_2 : \mathsf{vec}\ U\ n_2$. Thus $(\mathsf{Ind}\ n_1)\ (\lambda n.\lambda x.\lambda v.\mathsf{cons}(n+n_2)\ x\ v)\ l_2\ l_1 : \mathsf{vec}\ U\ (n_1 + n_2)$.

Theorem 4 (Associativity).
$\mu_v \vdash \forall U : *.\Pi(n_1, n_2, n_3 : \mathsf{Nat}).\Pi(v_1 : \mathsf{vec}\ U\ n_1, v_2 : \mathsf{vec}\ U\ n_2, v_3 : \mathsf{vec}\ U\ n_3).$
$(\mathsf{app}\ n_1\ (n_2 + n_3)\ v_1\ (\mathsf{app}\ n_2\ n_3\ v_2\ v_3) = \mathsf{app}\ (n_1 + n_2)\ n_3\ (\mathsf{app}\ n_1\ n_2\ v_1\ v_2)\ v_3$

Proof. Use $\mathsf{Ind}\ n_1$. We will not go through the proof here.

5 Metatheory

We first outline the erasure from $\mathbf{S}$ to $\mathbf{F}_\omega$ with positive recursive definitions. Then we conclude strong normalization for $\mathbf{S}$ by the strong normalization of $\mathbf{F}_\omega$ with positive recursive definitions. We also prove type preservation for $\mathbf{S}$, which involves *confluence analysis* (Section 5.2) and *morph analysis* (Section 5.3). All omitted proofs may be found in the extended version [11].

5.1 Strong Normalization

We prove strong normalization of $\mathbf{S}$ through the strong normalization of $\mathbf{F}_\omega$ with positive recursive definitions. We first define the syntax for $\mathbf{F}_\omega$ with positive recursive definitions. We work with kind-annotated types, for a tighter interpretation of types in the proof of Theorem 6.

Definition 8 (Syntax for $\mathbf{F}_\omega$ with positive definitions).

$$
\begin{array}{ll}
\textit{Terms } t & ::= \ x \mid \lambda x.t \mid tt' \\
\textit{Kinds } \kappa & ::= \ * \mid \kappa' \to \kappa \\
\textit{Types } T^\kappa & ::= \ X^\kappa \mid (\forall X^\kappa.T^*)^* \mid (T_1^* \to T_2^*)^* \mid (\lambda X^{\kappa_1}.T^{\kappa_2})^{\kappa_1 \to \kappa_2} \mid (T_1^{\kappa_1 \to \kappa_2} T_2^{\kappa_1})^{\kappa_2} \\
\textit{Context } \Gamma & ::= \ \cdot \mid \Gamma, x : T^\kappa \mid \Gamma, \mu \\
\textit{Definitions } \mu & ::= \ \{(x_i : S_i^\kappa) \mapsto t_i\}_{i \in N} \cup \{X_i^\kappa \mapsto T_i^\kappa\}_{i \in M} \\
\textit{Term definitions } \rho & ::= \ \{x_i \mapsto t_i\}_{i \in N}
\end{array}
$$

Note that for every $x \mapsto t, X^\kappa \mapsto T^\kappa \in \mu$, we require $\mathrm{FV}(t) = \emptyset$ and $\mathrm{FVar}(T^\kappa) \subseteq \{X^\kappa\}$; and the X^κ can only occur at the positive position in T^κ, no mutually recusive definitions are allowed. We elide the typing rules for space reason.

Definition 9 (Erasure for kinds). *We define a function F maps kinds in $\mathbf{S}$ to kinds in $\mathbf{F}_\omega$ with positive definitions.*

$F(*) \ := *$
$F(\Pi x : T.\kappa) \ := F(\kappa)$
$F(\Pi X : \kappa'.\kappa) \ := F(\kappa') \to F(\kappa)$

Definition 10 (Erasure relation). *We define relation $\Gamma \vdash T \triangleright T'^\kappa$ (intuitively, it means that type T can be erased to T'^κ under the context Γ), where T, Γ are types and context in $\mathbf{S}$, T'^κ is a type in $\mathbf{F}_\omega$ with positive definitions.*

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

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

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

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

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

Definition 11 (Erasure for Context). *We define relation $\Gamma \triangleright \Gamma'$ inductively.*

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

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

Theorem 5 (Erasure Theorem).

1. *If* $\Gamma \vdash T : \kappa$*, then there exists a* $T_a^{F(\kappa)}$ *such that* $\Gamma \vdash T \triangleright T_a^{F(\kappa)}$.
2. *If* $\Gamma \vdash t : T$ *and* $\Gamma \vdash \mathsf{wf}$*, then there exist* T_a^* *and* Γ' *such that* $\Gamma \vdash T \triangleright T_a^*$, $\Gamma \triangleright \Gamma'$ *and* $\Gamma' \vdash t : T_a^*$.

Now that we obtained an erasure from $\mathbf{S}$ to $\mathbf{F}_\omega$ with positive definitions. We continue to show latter is strongly normalizing. The development below is in $\mathbf{F}_\omega$ with positive definitions. Let $\mathfrak{R}_\rho$ be the set of all reducibility candidates[5]. Let σ be a mapping between type variable of kind κ to element of $\rho[\![\kappa]\!]$.

Definition 12.

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

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

Definition 13. *Let* $\rho = |\Gamma|$*, and* $\mathrm{FVar}(\Gamma)$ *be the set of free type variables in* Γ*. We define* $\sigma \in \rho[\![\Gamma]\!]$ *if* $\sigma(X^\kappa) \in \rho[\![\kappa]\!]$ *for undefined variable* X^κ*; and* $\sigma(X^\kappa) = \mathrm{lfp}(b \mapsto \rho[\![T^\kappa]\!]_{\sigma[b/X^\kappa]})$ *for* $b \in \rho[\![\kappa]\!]$ *if* $X^\kappa \mapsto T^\kappa \in \Gamma$.

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

Definition 14. *Let* $\rho = |\Gamma|$ *and* $\sigma \in \rho[\![\Gamma]\!]$*. We define the relation* $\delta \in \rho[\![\Gamma]\!]$ *inductively:*

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

Theorem 6 (Soundness theorem). *Let* $\rho = |\Gamma|$*. If* $\Gamma \vdash t : T^*$ *and* $\Gamma \vdash \mathsf{wf}$*, then for any* $\sigma, \delta \in \rho[\![\Gamma]\!]$*, we have* $\delta t \in \rho[\![T^*]\!]_\sigma$*, with* $\rho[\![T^*]\!]_\sigma \in \mathfrak{R}_\rho$.

Theorem 5 and 6 imply all the typable term in $\mathbf{S}$ is strongly normalizing.

5.2 Confluence Analysis

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

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

Definition 15.
$\Gamma \vdash T_1 \to_\iota T_2$ *if* $T_1 \equiv \iota x.T'$ *and* $T_2 \equiv [t/x]T'$ *for some fix term* t.

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

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

Lemma 1. $\to_\beta$ *is confluent.*

Definition 16 (Commutativity). *Let* $\to_1, \to_2$ *be two notions of reduction. Then* $\to_1$ *commutes with* $\to_2$ *iff* $\leftarrow_1 \cdot \to_2 \subseteq \to_1 \cdot \leftarrow_2$.

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

Lemma 2. $\to_\beta$ *commutes with* $\to_\iota$. *Thus* $\to_{\beta,\iota}$ *is confluent, where* $\to_{\beta,\iota} = \to_\beta \cup \to_\iota$.

Theorem 7 (ι-elimination). *If* $\Gamma \vdash \Pi x : T_1.T_2 =_{\beta,\iota} \Pi x : T_1'.T_2'$, *then* $\Gamma \vdash T_1 =_\beta T_1'$ *and* $\Gamma \vdash T_2 =_\beta T_2'$.

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

5.3 Morph Analysis

The methods of the previous section are not suitable for dealing with implicit polymorphism, since as a reduction relation, polymorphic instantiation is not confluent. For example, $\forall X : \kappa.X$ can be instantiated either to T or to $T \to T$. The only known syntactic method (to our knowledge) to deal with preservation proof for Curry-style System **F** is Barendregt's method [4]. We will extend his method to handle the instantiation of $\forall x : T.T'$.

Definition 17 (Morphing Relations).

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

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

Definition 18. $([\Gamma], T) =_{\beta,\iota} ([\Gamma], T')$ *if* $\Gamma \vdash T =_{\beta,\iota} T'$ *and* $\Gamma \vdash T : *$ *and* $\Gamma \vdash T' : *$.

The best way to understand the E, G mappings below is through understanding Lemmas 4 and 5. They give concrete demonstrations of how to *summarize* a sequence of morphing relations.

Definition 19.

$E(\forall X : \kappa.T) := E(T)$	$E(X) := X$	$E(\Pi x : T_1.T_2) := \Pi x : T_1.T_2$
$E(\lambda X.T) := \lambda X.T$	$E(T_1 T_2) := T_1 T_2$	$E(\forall x : T'.T) := \forall x : T'.T$
$E(\iota x.T) := \iota x.T$	$E(T\ t) := T\ t$	$E(\lambda x.T) := \lambda x.T$

Definition 20.

$G(\forall X : \kappa.T) := \forall X : \kappa.T$	$G(X) := X$	$G(\Pi x : T_1.T_2) := \Pi x : T_1.T_2$
$G(\lambda X.T) := \lambda X.T$	$G(T_1 T_2) := T_1 T_2$	$G(\forall x : T'.T) := G(T)$
$G(\iota x.T) := \iota x.T$	$G(T\ t) := T\ t$	$G(\lambda x.T) := \lambda x.T$

Lemma 3. $E([T'/X]T) \equiv [T''/X]E(T)$ *for some* T''*;* $G([t/x]T) \equiv [t/x]G(T)$.

Proof. By induction on the structure of T.

Lemma 4. *If* $([\Gamma], T) \to^*_{i,g} ([\Gamma'], T')$*, then there exists a type substitution* σ *such that* $\sigma E(T) \equiv E(T')$.

Proof. It suffices to consider $([\Gamma], T) \to_{i,g} ([\Gamma'], T')$. If $T' \equiv \forall X : \kappa.T$ and $\Gamma = \Gamma', X : \kappa$, then $E(T') \equiv E(T)$. If $T \equiv \forall X : \kappa.T_1$ and $T' \equiv [T''/X]T_1$ and $\Gamma = \Gamma'$, then $E(T) \equiv E(T_1)$. By Lemma 3, we know $E(T') \equiv E([T''/X]T_1) \equiv [T_2/X]E(T_1)$ for some T_2.

Lemma 5. *If* $([\Gamma], T) \to^*_{I,G} ([\Gamma'], T')$*, then there exists a term substitution* δ *such that* $\delta G(T) \equiv G(T')$.

Proof. It suffices to consider $([\Gamma], T) \to_{I,G} ([\Gamma'], T')$. If $T' \equiv \forall x : T_1.T$ and $\Gamma = \Gamma', x : T_1$, then $G(T') \equiv G(T)$. If $T \equiv \forall x : T_2.T_1$ and $T' \equiv [t/x]T_1$ and $\Gamma = \Gamma'$, then $E(T) \equiv E(T_1)$. By Lemma 3, we know $E(T') \equiv E([t/x]T_1) \equiv [t/x]E(T_1)$.

Lemma 6. *If* $([\Gamma], \Pi x : T_1.T_2) \to^*_{i,g} ([\Gamma'], \Pi x : T_1'.T_2')$*, then there exists a type substitution* σ *such that* $\sigma(\Pi x : T_1.T_2) \equiv \Pi x : T_1'.T_2'$.

Proof. By Lemma 4.

Lemma 7. *If* $([\Gamma], \Pi x : T_1.T_2) \to^*_{I,G} ([\Gamma'], \Pi x : T_1'.T_2')$*, then there exists a term substitution* δ *such that* $\delta(\Pi x : T_1.T_2) \equiv \Pi x : T_1'.T_2'$.

Proof. By Lemma 5.

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

Theorem 8 (Compatibility). *If* $([\Gamma], \Pi x : T_1.T_2) \to^*_{\iota,\beta,i,g,I,G} ([\Gamma'], \Pi x : T_1'.T_2')$, *then there exists a mixed substitution*[6] ϕ *such that* $([\Gamma], \phi(\Pi x : T_1.T_2)) =_{\iota,\beta} ([\Gamma], \Pi x : T_1'.T_2')$. *Thus* $\Gamma \vdash \phi T_1 =_\beta T_1'$ *and* $\Gamma \vdash \phi T_2 =_\beta T_2'$ *(by Theorem 7).*

Proof. By Lemma 7 and 6, making use of the fact that if $\Gamma \vdash t =_{\iota,\beta} t'$, then for any mixed substitution ϕ, we have $\Gamma \vdash \phi t =_{\iota,\beta} \phi t'$.

Theorem 9 (Type Preservation). *If* $\Gamma \vdash t : T$ *and* $\Gamma \vdash t \to_\beta t'$ *and* $\Gamma \vdash \mathsf{wf}$, *then* $\Gamma \vdash t' : T$.

6 $0 \neq 1$ in S

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

Definition 21. $\bot := \forall A : *.\forall x : A.\forall y : A.x = y$.

Theorem 10. *There is no term* t *such that* $\mu_c \vdash t : \bot$

Proof. Suppose $\mu_c \vdash t : \bot$. By the erasure theorem (Theorem 5) in Section 5.1, we have $F(\mu_c) \vdash t : \forall A : *.\forall C : *.C \to C$ in $\mathbf{F}_\omega$. We know that $\forall A : *.\forall C : *.C \to C$ is the singleton type[7], which is inhabited by $\lambda z.z$. This means $t \to^*_\beta \lambda z.z$ (the term reductions of $\mathbf{F}_\omega$ with let-bindings are the same as $\mathbf{S}$) and $\mu_c \vdash \lambda z.z : \bot$ in $\mathbf{S}$ (by type preservation, Theorem 9). Then we would have $\mu_c, A : *, x : A, y : A, C : A \to *, z : C\ x \vdash z : C\ y$. We know this derivation is impossible since $C\ x \ncong C\ y$ by the confluence of $\cong$.

Theorem 11. $\mu_c \vdash 0 = 1 \to \bot$.

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

7 Conclusion

We have revisited lambda encodings in type theory, and shown how a new self type construct $\iota x.T$ supports dependent eliminations with lambda encodings, including induction principles. We considered System $\mathbf{S}$, which incorporates self types together with implicit products and a restricted version of global positive recursive definition. The corresponding induction principles for Church- and Parigot-encoded datatypes are derivable in $\mathbf{S}$. By changing the notion of contradiction from explosion to equational inconsistency, we are able to show $0 \neq 1$ in both **CC** and $\mathbf{S}$. We proved type preservation, which is nontrivial for $\mathbf{S}$ since several rules are not syntax-directed. We also defined an erasure from $\mathbf{S}$ to $\mathbf{F}_\omega$ with positive definitions, and proved strong normalization of $\mathbf{S}$ by showing strong normalization of $\mathbf{F}_\omega$ with positive definitions. Future work includes further explorations of dependently typed lambda encodings for practical type theory. In particular, we would like to implement our system and carry out some case studies.

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

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

References

1. M. Abadi and L. Cardelli. A Theory of Primitive Objects - Second-Order Systems. In *European Symposium on Programming (ESOP)*, pages 1–25, 1994.
2. A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In G. Morrisett and T. Uustalu, editors, *International Conference on Functional Programming (ICFP)*, pages 185–196, 2013.
3. K.Y. Ahn, T. Sheard, M. Fiore, and A.M. Pitts. System Fi. In *Typed Lambda Calculi and Applications*, pages 15–30. 2013.
4. H. Barendregt. Lambda calculi with types, handbook of logic in computer science (vol. 2): background: computational structures, 1993.
5. B. Barras. Sets in coq, coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010.
6. V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.
7. A. Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. 1985.
8. T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, September 1989.
9. T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.
10. H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II*. 1972.
11. P. Fu and A. Stump. Self Types for Dependently Typed Lambda Encodings, 2014. Extended version available from `http://homepage.cs.uiowa.edu/~pfu/document/papers/rta-tlca.pdf`.
12. H. Geuvers. Inductive and Coinductive Types with Iteration and Recursion. In B. Nordstrom, K. Petersson, and G. Plotkin, editors, *Informal proceedings of the 1992 workshop on Types for Proofs and Programs*, pages 183–207, 1994.
13. H. Geuvers. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications (TLCA)*, pages 166–181, 2001.
14. E. Gimenez. *Un calcul de constructions infinies et son application a la verification de systemes communicants*. PhD thesis, 1996.
15. J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, 1972.
16. J. Hickey. Formal objects in type theory using very dependent types. In K. Bruce, editor, *In Foundations of Object Oriented Languages (FOOL) 3*, 1996.
17. A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. PhD thesis, PhD thesis, Université Paris 7, 2001.
18. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In L. Cardelli, editor, *17th European Conference on Object-Oriented Programming (ECOOP)*, pages 201–224, 2003.
19. M. Parigot. Programming with Proofs: A Second Order Type Theory. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP)*, pages 145–159, 1988.
20. D. Schepler. bijective function implies equal types is provably inconsistent with functional extensionality in coq. message to the Coq Club mailing list, December 12, 2013.
21. 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.