From Realizability to Induction via Dependent Intersection

Aaron Stump

Computer Science, MacLean Hall, The University of Iowa, Iowa City, IA, 52242

Abstract

In this paper, it is shown that induction is derivable in a type-assignment formulation of the second-order dependent type theory $\lambda P2$, extended with the implicit product type of Miquel, dependent intersection type of Kopylov, and heterogeneous equality type of McBride. The crucial idea is to use dependent intersections to internalize a result of Leivant's showing that Church-encoded data may be seen as realizing their own type correctness statements, under the Curry-Howard isomorphism.

1. Introduction

Constructive type theory has been proposed as a foundation for constructive mathematics, and has found numerous applications in Computer Science, thanks to the Curry-Howard correspondence between constructive logic and pure functional programming [23, 19, 12]. Pure Type Systems (PTSs) are one formalism for constructive type theory, based on pure lambda calculus [5]. PTSs have very compact syntax, reduction semantics, and typing rules, which is appealing from a foundational and metatheoretic perspective. Unfortunately, PTSs by themselves have not been found suitable as a true foundation for constructive type theory in practice, due to the lack of inductive types. At the introduction of the Calculus of Constructions (CC), an important impredicative PTS, induction was lacking [10]. This led the inventors of CC and their collaborators to extend the theory with a primitive notion of inductive types, resulting in the Calculus of Inductive Constructions (CIC), which is the core formalism of the prominent Coq computer-proof software [24, 33, 29, 11]. In 2001, this skepticism about induction in PTSs was solidified when Geuvers proved that induction is not derivable in second-order dependent type theory ($\lambda P2$), a subsystem of CC [16].

In this paper, we present an extension of a type-assignment formulation of $\lambda P2$, in which induction is derivable. The extension does not in any direct way correspond simply to adding primitive inductive types or induction principles. Rather, the extension strengthens the expressiveness of the dependent typing of $\lambda P2$, to take advantage of the computational power that is already present in impredicative type theory. The extension is with three constructs, all somewhat exotic but none new. The first is the implicit product $\forall\, x : A.\, B$ of Miquel, which allows one to generalize x of type A without introducing a λ-abstraction at the term level [26]. Second is the dependent intersection type of Kopylov [20]. Intersection types have been studied for many years in theoretical Computer Science, due to their strong connection with normalization properties (see [6] for a magisterial presentation). If a term t can be assigned types A and B, then it can also be assigned the type $A \cap B$. With dependent intersections, this is strengthened to: if a term t can be assigned types A and $[t/x]B$ (the substitution of t for x in B), then it can also be assigned the type $x : A \cap B$. In this paper, we will use the prefix notation $\iota\, x : A.\, B$, instead of Kopylov's $x : A \cap B$. The third construct in the extension is a heterogeneous equality type, as proposed by McBride [25]. In type theory,

Email address: `aaron-stump@uiowa.edu` (Aaron Stump)
This is a draft paper currently under review. The version you are reading is likely to be revised before final publication.

$$\begin{array}{lll}
\textit{terms } t & ::= & x \mid t\ t' \mid \lambda\, x.\, t \\
\textit{types } T & ::= & X \mid \forall\, X\!:\!\kappa.\, T \mid \Pi\, x\!:\!T.\, T' \mid \lambda\, x\!:\!T.\, T' \mid T\ t \mid \forall\, x\!:\!T.\, T' \mid \iota\, x\!:\!T.\, T' \mid t \simeq t' \\
\textit{kinds } \kappa & ::= & \star \mid \Pi\, x\!:\!T.\, \kappa \\
\textit{contexts } \Gamma & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, X : \kappa
\end{array}$$

Figure 1: Sytax for $\iota\lambda P2$

homogeneous equality types allow expression of equality between terms x and y both of some common type A. With heterogeneous equality, the requirement of a common type is dropped, and we may form equalities between terms x and y of different types. While all three constructs are necessary for the derivation given of induction, the dependent intersections are most central to the construction, and so we will denote the resulting system $\iota\lambda P2$.

The centrality of dependent intersection for induction in $\iota\lambda P2$ is due to its role in internalizing a crucial realizability result of Leivant [22]. He observed that the proofs that data encoded as pure lambda terms using the well-known Church encoding satisfy their typing laws can be identified with those data themselves. In other words, Church-encoded numbers realize their own typings. This remarkable observation is the key to the construction in this paper.

Section 2 defines the $\iota\lambda P2$ type theory. This is a type-assignment system, and as it stands, unsuitable for use as a type-checking algorithm. In Section 3 we propose a scheme for annotating terms with sufficient information to make typing essentially subject-directed. Section 4 gives a definition of the type of natural numbers and, using the notation of annotated $\iota\lambda P2$, constructs an inhabitant of the statement of natural-number induction for this type. This construction has been checked in a prototype implementation of annotated $\iota\lambda P2$. Section 5 gives a realizability semantics for $\iota\lambda P2$, from which the existence of an uninhabited type is an easy corollary. This confirms that the addition of the three constructs of $\iota\lambda P2$ to $\lambda P2$ has not led to an inconsistent theory. Section 6 discusses some of the consequences of the result, and related work. We conclude in Section 7 with future directions for strengthening $\iota\lambda P2$ to a full-featured dependent type theory.

2. The $\iota\lambda P2$ Type Theory

The syntax for $\iota\lambda P2$ is given in Figure 1, where we use x for term variables and X for type variables. We follow standard conventions for syntactic concepts like variable scoping, capture-avoiding substitution, α-equivalence, etc. $\forall\, X\!:\!T.$ is impredicative universal quantification over types as in $\lambda 2$ (System F). $\Pi\, x\!:\!T.\, T'$ is the dependent function type, which is also written $T \to T'$ when $x \notin FV(T')$ (the set of free variables of T'). $\lambda\, x\!:\!T.\, T'$ is for type-level λ-abstraction over terms, and $T\ t$ is the corresponding application. To this point in the syntax for types in Figure 1, we have just the types of $\lambda P2$. The extensions come next.

$\forall\, x\!:\!T.\, T'$ is the implicit product of Miquel: intuitively, it is the type for terms t which may be assigned type T' for any value for x of type T [26]. $\iota\, x\!:\!T.\, T'$ is notation for Kopylov's dependent intersection type. This is a binding notation, where the scope of bound variable x is T'. $t \simeq t'$ is notation for McBride's heterogeneous equality type. Note that $\iota\lambda P2$, just like $\lambda P2$, does not allow the formation of type-level λ-abstractions over types. Figure 1 also includes the syntax for a simple language of kinds κ, which classify types; and of typing contexts Γ, which record assumptions about free term- and type-level variables (x and X, respectively).

The type system of $\iota\lambda P2$ comprises the mutually inductive definition of three judgments:

- $\Gamma \vdash \kappa$ expresses that kind κ is well-formed in context Γ (Figure 2),
- $\Gamma \vdash T : \kappa$ expresses that type T has kind κ in context Γ (Figure 3), and
- $\Gamma \vdash t : T$ expresses that term t may be assigned type T in context Γ (Figure 4).

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

Figure 2: Rules for judging that a kind is well-formed in context ($\Gamma \vdash \kappa$)

$$\frac{\Gamma \vdash \kappa}{\Gamma, X : \kappa \vdash X : \kappa} \quad \frac{\Gamma \vdash \kappa' \quad \Gamma \vdash T : \kappa}{\Gamma, X : \kappa' \vdash T : \kappa'} \quad \frac{\Gamma \vdash T : \star \quad \Gamma \vdash T' : \kappa}{\Gamma, x : T \vdash T' : \kappa}$$

$$\frac{\Gamma, X : \kappa \vdash T : \star}{\Gamma \vdash \forall\, X{:}\kappa.\, T : \star} \quad \frac{\Gamma, x : T \vdash T' : \star}{\Gamma \vdash \Pi\, x{:}T.\, T' : \star} \quad \frac{\Gamma, x : T \vdash T' : \kappa}{\Gamma \vdash \lambda\, x{:}T.\, T' : \Pi\, x{:}T.\, \kappa}$$

$$\frac{\Gamma \vdash T : \Pi\, x{:}T'.\, \kappa \quad \Gamma \vdash t : T'}{\Gamma \vdash T\ t : [t/x]\kappa} \quad \frac{\Gamma, x : T \vdash T' : \star}{\Gamma \vdash \forall\, x{:}T.\, T' : \star} \quad \frac{\Gamma, x : T \vdash T' : \star}{\Gamma \vdash \iota\, x{:}T.\, T' : \star}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \simeq t' : \star}$$

Figure 3: Rules for judging that a type has a kind in context ($\Gamma \vdash T : \kappa$)

The second and third rules in each figure are weakening rules. The last rule of Figure 4 is a conversion rule, for changing a type to a β-equivalent one. The notation $T =_\beta T'$ refers to standard β-equivalence, including both term- and type-level β-conversion, of the classifiers in question. We do not need a similar kind-level conversion rule for the derivation of induction below, so this is omitted. Note that definitional equalities are to be distinguished from the heterogeneous equality type $t \simeq t'$. The elimination rule for $t \simeq t'$ makes it a true type-theoretic equality, in the sense of being substitutive. It may seem strange to substitute a term of one type for a term of another, but it is important to keep in mind that with heterogeneous equality, the introduction rule ensures that the only equalities that can be proved are ones between terms of equal types. This is the bottom-left rule of Figure 4; note that it arbitrarily uses $\lambda\, x.\, x$ as the proof of the trivial heterogeneous equality. So the difference in types for the sides of a heterogeneous equality is allowed in stating equalities, but not in proving them. This makes the relaxation to different types somewhat illusory; indeed, it led McBride humorously to dub these types "John Major" equalities, which promise greater equality but in reality are just as exclusionary as previously (with homogeneous equality) [25].

3. Annotated $\iota\lambda P2$

The type-assignment formulation of $\iota\lambda P2$ presented in the previous section is not subject-directed: many rules do not change the subject of typing when passing from conclusion to premises. This means that, as usual with type-assignment systems, it is not obvious how to use the system as a type-checking algorithm. To make the derivation of induction in Section 4 more informative, this section presents an annotated version of $\iota\lambda P2$, with subject-directed versions of the term-typing rules, based on local type inference (also known as bidirectional type checking) [30]. The relation $\Gamma \vdash t : T$ of unannotated $\iota\lambda P2$ is replaced in the annotated version with two relations: $\Gamma \vdash t \Leftarrow T$ and $\Gamma \vdash t \Rightarrow T$. In the former, Γ, t, and T are inputs; in the latter, Γ and t are inputs, and T is output. The syntax for annotated terms is given in Figure 5; the constructs will be explained below, with the typing rules for annotated terms. The syntax for types and kinds is exactly the same, but all references to terms t should now be understood to be to annotated terms. Annotated terms erase to unannotated ones as shown in Figure 6. We extend this function to types, kinds, and contexts in the obvious way, by applying the eraser function of Figure 6 to any terms contained in expressions of those other forms.

The rules for judging kinds well-formed are unchanged in annotated $\iota\lambda P2$ from unannotated $\iota\lambda P2$. The

$$\frac{\Gamma \vdash T : \star}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma \vdash \kappa \quad \Gamma \vdash t : T}{\Gamma, X : \kappa \vdash t : T} \qquad \frac{\Gamma \vdash T' : \star \quad \Gamma \vdash t : T}{\Gamma, x : T' \vdash t : T}$$

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

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

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash t : [t/x]T'}{\Gamma \vdash t : \iota\, x{:}T.\, T'} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t : T} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t : [t/x]T'}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \lambda\, x.\, x : t \simeq t} \qquad \frac{\Gamma \vdash t' : t_1 \simeq t_2 \quad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash t : [t_2/x]T} \qquad \frac{T =_\beta T' \quad \Gamma \vdash T : \star \quad \Gamma \vdash t : T'}{\Gamma \vdash t : T}$$

Figure 4: Rules for judging that a term can be assigned a type in context ($\Gamma \vdash t : T$)

$$\textit{annotated terms } t \quad ::= \quad x \mid \lambda\, x.\, t \mid t\ t' \mid \Lambda\, X.\, t \mid t \cdot T \mid \Lambda\, x.\, t \mid t \text{ -}t' \mid [t, t'] \mid t.1 \mid t.2 \mid \beta \mid \rho\ t \text{ - } t'$$

Figure 5: The syntax for annotated terms

kinding rules are also identical to the unannotated ones, with the exception of the rule for kinding equations $t \simeq t'$ and the rule for kinding type-level applications $T\ t$. These rules are to be replaced by the ones in Figure 7.

The typing rules for annotated terms are in Figure 8. In a few rules we use $\Leftrightarrow$ as a meta-variable ranging over $\{\Leftarrow, \Rightarrow\}$. The rules are still not fully algorithmic: the use of weakening rules and the conversion rules are not subject-directed. The rule for ρ-terms is also nondeterministic, because it is not clear which instances of t_1 to rewrite to t_2 (it is sufficient for our purposes below just to rewrite them all). The weakening and conversion rules can be incorporated into a fully algorithmic version of the typing and kinding rules in a standard way, and so to avoid unnecessary technicalities, we will not carry out this step here.

The rules of Figure 8 are in one-one correspondence with the rules of Figure 4 above. With the exceptions just noted (weakening and conversion), the rules are now subject-directed: there is a distinct form of annotated term in the conclusion of all the rules. Where the unannotated rules just use $\Gamma \vdash t : T$ in their premises and conclusions, the annotated rules refine this to $\Gamma \vdash t \Leftarrow T$ in some cases, and $\Gamma \vdash t \Rightarrow T$ in others (and allow either possibility for weakening and conversion rules). For the type form $\forall\, X{:}\kappa.\, T$, we have annotation forms $\Lambda\, X.\, t$ and $t \cdot T$, for introduction and elimination respectively. Similarly, for $\forall\, x{:}T'.\, T$, we have $\Lambda\, x.\, t$ and t -t' (so t' is an erased, or implicit, argument). For the dependent intersection type, we have constructs $[t, t']$, $t.1$, and $t.2$, which look like constructs for ordered pairs, but here should be interpreted as operating on different views of the same term t. So $[t, t']$ has a dependent intersection type $\iota\, x : T'.\, T$ iff t and t' are different annotations, corresponding to different type-assignments, for the same unannotated term $|t|$ (hence the requirement that $|t| = |t'|$ in the premise). Finally, for the heterogeneous equality type $t \simeq t'$, we have annotated terms β for introduction and $\rho\ t$ - t' for elimination. The former allows us to prove $t \simeq t'$ when t and t' erase to the same unannotated term. Combined with the conversion rule, this allows us to prove terms equal if their erasures are convertible. The latter construct allows us to rewrite t_1 to t_2 in the type of t', when the type of t is $t_1 \simeq t_2$.

The annotated version of $\iota\lambda P2$ has been designed to be subject-directed (with the exceptions noted above), and to enable completely routine validation of the following soundness theorem:

Theorem 1. *If $\Gamma \vdash t \Leftarrow T$ or $\Gamma \vdash t \Rightarrow T$ (in annotated $\iota\lambda P2$), then $|\Gamma| \vdash |t| : |T|$ (in unannotated $\iota\lambda P2$).*

$$
\begin{array}{lcl}
|x| & = & x \\
|\lambda\, x.\, t| & = & \lambda\, x.\, |t| \\
|t\ t'| & = & |t|\ |t'| \\
|t \cdot T| & = & |t| \\
|\Lambda\, x.\, t| & = & |t| \\
|t\ \text{-}t'| & = & |t| \\
|[t, t']| & = & |t| \\
|t.1| & = & |t| \\
|t.2| & = & |t| \\
|\beta| & = & \lambda\, x.\, x \\
|\rho\ t\ \text{-}\ t'| & = & |t'|
\end{array}
$$

Figure 6: Eraser function for annotated terms

$$
\frac{\Gamma \vdash T : \Pi\, x{:}T'.\, \kappa \quad \Gamma \vdash t \Leftarrow T'}{\Gamma \vdash T\ t : [t/x]\kappa}
\qquad
\frac{\Gamma \vdash t \Rightarrow T \quad \Gamma \vdash t' \Rightarrow T'}{\Gamma \vdash t \simeq t' : \star}
$$

Figure 7: Modified kinding rules referring to annotated terms

$$
\frac{\Gamma \vdash T : \star}{\Gamma, x : T \vdash x \Leftrightarrow T}
\qquad
\frac{\Gamma \vdash \kappa \quad \Gamma \vdash t \Leftrightarrow T}{\Gamma, X : \kappa \vdash t \Leftrightarrow T}
$$

$$
\frac{\Gamma \vdash T' : \star \quad \Gamma \vdash t \Leftrightarrow T}{\Gamma, x : T' \vdash t \Leftrightarrow T}
\qquad
\frac{\Gamma, x : T \vdash t \Leftarrow T'}{\Gamma \vdash \lambda\, x.\, t \Leftarrow \Pi\, x{:}T.\, T'}
$$

$$
\frac{\Gamma \vdash t \Rightarrow \Pi\, x{:}T'.\, T \quad \Gamma \vdash t' \Leftarrow T'}{\Gamma \vdash t\ t' \Rightarrow [t'/x]T}
\qquad
\frac{\Gamma, X : \kappa \vdash t \Leftarrow T}{\Gamma \vdash \Lambda\, X.\, t \Leftarrow \forall\, X{:}\kappa.\, T}
$$

$$
\frac{\Gamma \vdash t \Rightarrow \forall\, X{:}\kappa.\, T \quad \Gamma \vdash T' \Leftarrow \kappa}{\Gamma \vdash t \cdot T' \Rightarrow [T'/X]T}
\qquad
\frac{\Gamma, X : T' \vdash t \Leftarrow T}{\Gamma \vdash \Lambda\, x.\, t \Leftarrow \forall\, x{:}T'.\, T}
$$

$$
\frac{\Gamma \vdash t \Rightarrow \forall\, x{:}T'.\, T \quad \Gamma \vdash t' \Leftarrow T'}{\Gamma \vdash t\ \text{-}t' \Rightarrow [t'/X]T}
\qquad
\frac{\Gamma \vdash t \Leftarrow T \quad \Gamma \vdash t' \Leftarrow [t/x]T' \quad |t| = |t'|}{\Gamma \vdash [t, t'] \Leftarrow \iota\, x{:}T.\, T'}
$$

$$
\frac{\Gamma \vdash t \Rightarrow \iota\, x{:}T.\, T'}{\Gamma \vdash t.1 \Rightarrow T}
\qquad
\frac{\Gamma \vdash t \Rightarrow \iota\, x{:}T.\, T'}{\Gamma \vdash t.2 \Rightarrow [t.1/x]T'}
$$

$$
\frac{\Gamma \vdash t \Rightarrow T \quad |t| = |t'|}{\Gamma \vdash \beta \Leftarrow t \simeq t'}
\qquad
\frac{\Gamma \vdash t' \Rightarrow t_1 \simeq t_2 \quad \Gamma \vdash t \Leftrightarrow [t_1/x]T}{\Gamma \vdash \rho\ t'\ \text{-}\ t \Leftrightarrow [t_2/x]T}
$$

$$
\frac{|T| =_\beta |T'| \quad \Gamma \vdash T : \star \quad \Gamma \vdash t \Leftrightarrow T'}{\Gamma \vdash t \Leftrightarrow T}
$$

Figure 8: Bidirectional typing rules for annotated terms

```
cNat ⇐ ⋆ = ∀ X : ⋆ . (X → X) → X → X .
cZ ⇐ cNat = Λ X . λ s . λ z . z .
cS ⇐ cNat → cNat = λ n . Λ X . λ s . λ z . s (n · X s z) .
```

Figure 9: Definition of Church-encoded natural numbers and their constructors

4. Deriving Induction in $\iota\lambda P2$

The central idea for the derivation of induction in $\iota\lambda P2$ is, as mentioned above, to internalize a realizability result of Leivant's about Church-encoded natural numbers. Let us review this here briefly, for the case of natural numbers. The setting is a natural-deduction formulation of (single-sorted) second-order logic. Suppose we have a primitive unary function S and constant 0, and define a predicate N as follows, where $\forall R^1$ denotes universal quantification over unary predicate R, and $\forall z$ just first-order quantification:

$$N\ x \ = \ \forall R^1.(\forall z.R\ z \to R\ (S\ z)) \to R\ 0 \ \to R\ x$$

Then for any term n constructed from S and 0, the normal-form natural-deduction proof in second-order logic of the formula $N\ n$ may be identified, under the Curry-Howard isomorphism, with the Church encoding of n (more precisely, with a type-annotated version of this term). For the proof must assume arbitrary unary predicate R, and then make assumptions s and z of the antecedents of the implication. Then, in essence, s must be applied n times to z to prove $R\ n$. Thus the proof can be seen as a type-annotated version of $\lambda\, s.\, \lambda\, z.\, \underbrace{s\ \cdots\ (s}_{n}\ z)$ – and this is indeed the Church encoding of n.

4.1. *The type cNat*

We internalize Leivant's observation by first defining a type *cNat* of Church-encoded natural numbers, and their constructors *cZ* (zero) and *cS* (successor), in the usual way (due to Fortune, Leivant, and O'Donnell [15]). This is done in Figure 9, using the notation for annotated $\iota\lambda P2$ presented in the previous section. We write

```
symbol ⇐ classifier = definiens
```

to indicate a global definition of `symbol` with the given `classifier` (type or kind) by the given `definiens`. Note that the code in this figure and the subsequent ones has been checked by a prototype implementation of $\iota\lambda P2$, and copied from the source file verbatim.

4.2. *The type Nat*

Next we define the type *Nat* as shown in Figure 10. This type is the crucial internalization of Leivant's observation. We are defining "true" natural numbers to be those terms which are both Church-encoded natural numbers x and also realizers of the statement of induction specialized to x. Kopylov's dependent intersection type (the ι-type in Figure 10) is critical here, to allow us to express that x realizes its own induction principle.

We are using an implicit product type in the statement of the successor (step) case of induction, namely `∀ x : cNat . Q x → Q (cS x)`. This use of implicit products allows the erased normal forms inhabiting the second part of the ι-type defining `Nat` to match exactly those of `cNat`. If we used a regular dependent product type here instead, then in normal forms for that second part, there would be an extra λ-abstraction which the normal forms of type `cNat` lack.

Given the definition of *Nat* in Figure 10, how do we define constructors Z and S? This is shown in Figure 11. We are using the annotated terms $[t, t']$ and $t.1$ and $t.2$ for introducing and eliminating ι-types, respectively.

```
Nat ⇐ ⋆ = ι x : cNat . ∀ Q : cNat → ⋆ . (∀ x : cNat . Q x → Q (cS x)) → Q cZ → Q x.
```

Figure 10: Definition of *Nat* type

```
Z ⇐ Nat = [ cZ , Λ X . λ s . λ z . z ] .
S ⇐ Nat → Nat = λ n . [ cS n.1 , Λ P . λ s . λ z . s -n.1 (n · P s z) ] .
```

Figure 11: Definition of constructors for *Nat*

The typing rule (Figure 8) for the first of these, combined with conversion, requires that t and t' erase to convertible unannotated terms. We can confirm this for the components of the ι-introduction in the definition of Z: they both erase to `λ s . λ z . z`. For the definition of S: recall from Figure 6 that $n.1$, used in this definition, erases to n; also `s -n.1` erases to just `s`. So the erasure of the two components of the ι-introduction for S both are convertible to

```
λ s . λ z . s (n s z)
```

4.3. Proving induction for Nat

We are ready now to derive induction for type *Nat*. We will construct an inhabitant of the type

```
∀ Q : Nat → ⋆ . (∀ x : Nat . Q x → Q (S x)) → Q Z → Π x : Nat . Q x
```

Informally, here is the basic idea. We take in arguments Q, s, and z, corresponding to the first three abstractions in that type; and then an x of type *Nat*. We will use $x.2$ to construct x' of type *Nat*, together with a proof of $Q\ x'$. This is easily done, just by an iteration of the S constructor of *Nat* starting with Z, alongside an iteration of s starting from z. The former iteration builds the value x' of type *Nat*, and the latter builds the proof of $Q\ x'$. From outside the system, we can observe that if we iterate S starting from Z the same number of times as the number represented by $x.2$, then we will get a Church-encoded number also representing $x.2$. How do we internalize this observation? We use McBride's heterogeneous equality to state that the x' (of type *Nat*) we are constructing is equal to $x.1$ (of type *cNat*) for which $x.2$ is allowing us to perform a dependent elimination.

Let us now work through the details of the derivation in $\iota\lambda P2$, shown in Figure 12. We are defining `Ind` whose type is the induction principle for *Nat*. The derivation of this principle begins after the equals sign, with Λ `Q`. We first take the following inputs:

- `Q` of type `Nat → ⋆`
- `s` of type `∀ x : Nat . Q x → Q (S x)`
- `z` of type `Q Z`
- `x` of type `Nat`

```
Ind ⇐ ∀ Q : Nat → ⋆ . (∀ x : Nat . Q x → Q (S x)) → Q Z → Π x : Nat . Q x =
  Λ Q . λ s . λ z . λ x .
    x.2 ·
     (λ x : cNat . ∀ X : ⋆ . (Π x' : Nat . (x ≃ x') → Q x' → X) → X)
     (Λ x' . λ ih . Λ X . λ c .
        ih · X (λ x'' . λ e . λ u . c (S x'') (ρ e - β) (s -x'' u)))
     (Λ X . λ c . c Z β z)
     · (Q x)
     (λ x' . λ e . λ u . ρ e - u).
```

Figure 12: Derivation of induction in $\iota\lambda P2$

We are obliged now to produce a value of type `Q x`. We do this following the plan described informally above. We begin with an elimination of `x.2`. From the definition of `Nat` and the annotated typing rule for `x.2`, the type of `x.2` is:

```
∀ Q : (cNat → ⋆) . ((∀ x' : cNat . ((Q x') → (Q (cS x')))) → (Q cZ) → (Q x.1))
```

So the first thing we must do when performing an elimination with `x.2` is to supply the instace of `Q` (what is sometimes called the **motive** [25]), which in the code of Figure 12 is given as the following type (fourth line from the top of the code):

```
λ x : cNat . ∀ X : ⋆ . (Π x' : Nat . (x ≃ x') → Q x' → X) → X
```

We are indicating that we want to compute a value of the following type (let us call it R), where `x` in the line above has been replaced by `x.1`, by dependent iteration:

```
∀ X : ⋆ . ((Π x' : Nat . ((x.1 ≃ x') → (Q x') → X)) → X)
```

This is the type for a Church-encoded triple consisting of

- `x'` of type `Nat`,
- a proof of `x.1 ≃ x'`, and
- a proof of `Q x'`

Once we have computed this pair, we will be able to extract a proof of `Q x`, as done in the bottom two lines of Figure 12: we instantiate the type variable `X` in the type R with `Q x`, and then return the third component of the triple, casting `Q x'` to `Q x.1` using the second component. Since `Q x.1` erases to `Q x`, the conversion rule allows us to view the result as having type `Q x`, as required.

We must look now at the successor and zero cases of the dependent elimination of `x.2`, to complete our detailed examination of the code of Figure 12. Since the zero case is much shorter, let us look at it first. It is on the seventh line from the top of Figure 12:

```
Λ X . λ c . c Z β z
```

We take in inputs `X` of kind ⋆ and `c` of type

```
Π x' : Nat . ((cZ ≃ x') → (Q x') → X)
```

We are obligated to produce a result of type `X`, which can only be done by applying `c`. This we do, to the triple corresponding to `cZ`:

- `Z` of type `Nat`,
- β which proves that the erasure of `cZ` is β-equivalent to the erasure of `Z`, and
- `z` which proves `Q Z`

Now let us consider the successor case, in the fifth and sixth lines of the figure:

```
Λ x' . λ ih . Λ X . λ c .
  ih · X (λ x'' . λ e . λ u . c (S x'') (ρ e - β) (s -x'' u))
```

We are first taking in the following inputs (their types are determined by the type of `x.2`, given the motive we have supplied):

- `x'` of type `cNat`
- `ih` of type `∀ X : ⋆ . ((Π x'' : Nat . ((x' ≃ x'') → (Q x'') → X)) → X)`
- `X` of kind ⋆
- `c` of type `Π x'' : Nat . (((cS x') ≃ x'') → (Q x'') → X)`

Intuitively, the `ih` is the triple corresponding to `x'`, and we must produce a triple corresponding to `cS x'`. For that, we are obliged to return now something of type `X`, by applying `c` to the components of that triple for `cS x'`. We first access the components of the triple for `x'` by eliminating `ih` (sixth line from the top of Figure 12):

```
ih · X (λ x'' . λ e . λ u . c (S x'') (ρ e - β) (s -x'' u))
```

We are trying to produce a value of type `X`, so that is the first argument to `ih`. The components of the triple are then made available to us as

- `x''` of type `Nat`,
- `e` of type `x' ≃ x''`, and
- `u` of type `Q x''`.

We pass now to `c` the components of the new triple (the one for `cS x'`):

- `S x''` of type `Nat`,
- `ρ e - β` of type `(cS x') ≃ (S x'')`, and
- `s -x'' u` of type `Q (S x'')`.

Let us look carefully at the typings for each of these components. First, since `x''` is of type `Nat`, we have `S x''` also of type `Nat`, since `S` is of type `Nat → Nat`. Next, to prove `(cS x') ≃ (S x'')`, it suffices to rewrite `x'` to `x''` and then check that `cS x''` is β-equivalent to `S x''`. From the definitions of `S` and `cS`, and of erasure on the construct $[t, t']$, this is the case. Finally, to apply `s` we must specify an erased argument of type `Nat`. This is `x''`. We must also supply a proof of `Q x''`, and this is `u`. This completes the detailed examination of the code in Figure 12.

5. Realizability Semantics for $\iota\lambda P2$

In this section, we develop a realizability semantics for $\iota\lambda P2$ types, which we then use to show logical consistency of $\iota\lambda P2$. We use a simplified form of a similar semantics proposed in previous work [31]. The semantics uses set-theoretic partial functions for higher-kinded types. An application of such a function is undefined if the argument is not in the domain of the partial function. Any meta-level expressions, including formulas, which contain undefined subexpressions are undefined themselves. We write $A \to B$ for the set of meta-level total functions from set A to set B. We write $(x \in A \mapsto b)$ for the (meta-level) function mapping input x in the set A to b.

Let $\mathcal{L}$ be the set of closed lambda abstractions. We will write $\rightsquigarrow$ for (full) β-reduction. We also write $=_{c\beta}$ for standard β-equivalence restricted to closed terms, and $[t]_{c\beta}$ for the set $\{t' \mid t =_{c\beta} t'\}$. The latter operation is extended to sets S of terms by writing $[S]_{c\beta}$ for $\{[t]_{c\beta} \mid t \in S\}$.

Definition 2 (Reducibility candidates)**.** $\mathcal{R} := \{[S]_{c\beta} \mid S \subseteq \mathcal{L}\}$.

A reducibility candidate (element of $\mathcal{R}$) is a set of $c\beta$-equivalence classes of λ-abstractions. We will make use of a **choice function** ζ: given any set E of terms, ζ returns a λ-abstraction if E contains one, and is undefined otherwise.

Lemma 3 ($\mathcal{R}$ is a complete lattice)**.** *The set $\mathcal{R}$ ordered by subset forms a complete lattice, with greatest element $[\mathcal{L}]_{c\beta}$, least element $\emptyset$, and greatest lower bound of a nonempty set of elements given by intersection.*

Figure 13 defines our semantics for types and kinds, by mutual structural recursion. The semantic functions take arguments σ and ρ, in addition to the type or kind to interpret. We require that σ maps term variables to terms, and ρ maps type variables to sets. The interpretations of types and kinds are then also sets. The meaning of a type can be empty, and so in interpreting $\forall\, x : T. T'$ we must take the intersection using $\cap_\star$,

$$
\begin{array}{lcl}
[\![X]\!]_{\sigma,\rho} & = & \rho(X) \\
[\![\Pi x : T_1.T_2]\!]_{\sigma,\rho} & = & [\{\lambda x.t \mid \forall E \in [\![T_1]\!]_{\sigma,\rho}.\ [[\zeta(E)/x]t]_{c\beta} \in [\![T_2]\!]_{\sigma[x\mapsto\zeta(E)],\rho}\}]_{c\beta} \\
[\![\forall X : \kappa.T]\!]_{\sigma,\rho} & = & \cap\{[\![T]\!]_{\sigma,\rho[X\mapsto S]} \mid S \in [\![\kappa]\!]_{\sigma,\rho}\} \\
[\![\forall x : T.T']\!]_{\sigma,\rho} & = & \cap_\star\{[\![T']\!]_{\sigma[x\mapsto\zeta(E)],\rho} \mid E \in [\![T]\!]_{\sigma,\rho}\} \\
[\![\iota x : T.T']\!]_{\sigma,\rho} & = & \{E \in [\![T]\!]_{\sigma,\rho} \mid E \in [\![T']\!]_{\sigma[x\mapsto\zeta(E)],\rho}\} \\
[\![\lambda x : T.T']\!]_{\sigma,\rho} & = & (E \in [\![T]\!]_{\sigma,\rho} \mapsto [\![T']\!]_{\sigma[x\mapsto\zeta(E)],\rho}) \\
[\![T\ t]\!]_{\sigma,\rho} & = & [\![T]\!]_{\sigma,\rho}([(\sigma t)]_{c\beta}) \\
[\![t \simeq t']\!]_{\sigma,\rho} & = & \sigma t =_{c\beta} \sigma t' \\
[\![\star]\!]_{\sigma,\rho} & = & \mathcal{R} \\
[\![\Pi x : T.\kappa]\!]_{\sigma,\rho} & = & (E \in [\![T]\!]_{\sigma,\rho} \to [\![\kappa]\!]_{\sigma[x\mapsto\zeta(E)],\rho}),\ \text{if } [\![T]\!]_{\sigma,\rho} \in \mathcal{R} \\
\cap_\star S & = & \begin{cases} \cap S, & \text{if } S \neq \emptyset \\ [\mathcal{L}]_{c\beta}, & \text{otherwise} \end{cases}
\end{array}
$$

Figure 13: Semantics for types and kinds

$$
\begin{array}{lcl}
(\sigma \uplus [x \mapsto t], \rho) \in [\![\Gamma, x : T]\!] & \Leftrightarrow & (\sigma,\rho) \in [\![\Gamma]\!] \ \wedge\ [\![T]\!]_{\sigma,\rho} \in \mathcal{R} \ \wedge\ [t]_{c\beta} \in [\![T]\!]_{\sigma,\rho} \\
(\sigma, \rho \uplus [X \mapsto S]) \in [\![\Gamma, X : \kappa]\!] & \Leftrightarrow & (\sigma,\rho) \in [\![\Gamma]\!] \ \wedge\ S \in [\![\kappa]\!]_{\sigma,\rho} \\
(\emptyset,\emptyset) \in [\![\cdot]\!] & &
\end{array}
$$

Figure 14: Semantics of typing contexts Γ

defined at the end of Figure 13, which returns the top element of $\mathcal{R}$ if the interpretation of T is empty. The meaning of a kind cannot be empty, however, so we do not need to worry about this situation when interpreting $\forall X : \kappa.T$. For the semantics of $\Pi x : T.\kappa$, if $[\![T]\!]_{\sigma,\rho} \notin \mathcal{R}$, then the meaning of the Π-kind is undefined.

Figure 14 defines a semantics for typing contexts, in a standard way. Disjoint union is written $\uplus$. With this, we can prove the main theorem by mutual induction on the assumed derivations:

Theorem 4 (Soundness of typing and kinding)**.** *If* $(\sigma,\rho) \in [\![\Gamma]\!]$*, then*

1. *If* $\Gamma \vdash \kappa : \Box$*, then* $[\![\kappa]\!]_{\sigma,\rho}$ *is defined.*
2. *If* $\Gamma \vdash T : \kappa$*, then* $[\![T]\!]_{\sigma,\rho} \in [\![\kappa]\!]_{\sigma,\rho}$*.*
3. *If* $\Gamma \vdash t : T$*, then* $[\sigma t]_{c\beta} \in [\![T]\!]_{\sigma,\rho}$ *and* $[\![T]\!]_{\sigma,\rho} \in \mathcal{R}$*.*

For the conversion case of the proof of Theorem 4, it is crucial that our semantics has identical interpretations of convertible types:

Lemma 5. *If* $\Gamma \vdash T =_\beta T'$ *and for some kinds* κ *and* κ'*,* $[\![T]\!]_{\sigma,\rho} \in [\![\kappa]\!]_{\sigma,\rho}$ *and* $[\![T']\!]_{\sigma,\rho} \in [\![\kappa']\!]_{\sigma,\rho}$*, then* $[\![T]\!]_{\sigma,\rho} = [\![T']\!]_{\sigma,\rho}$*.*

From soundness of the typing and kinding rules for the semantics, we obtain:

Corollary 6 (Logical consistency)**.** *There is no derivation of* $\cdot \vdash t : \forall X : \star.X$*, for any term* t*.*

Proof. By Theorem 4 part (3) and the semantics of $\forall$-types, if $\cdot \vdash t : \forall X : \star.X$ is derivable, then $t \in \cap\mathcal{R}$. But $\cap\mathcal{R}$ is empty since $\emptyset \in \mathcal{R}$ by Lemma 3. $\Box$

Thus, $\iota\lambda P2$ is sound for use as a logic under the Curry-Howard isomorphism. This provides evidence that adding implicit products, dependent intersections, and heterogeneous equalities to $\lambda P2$ has not spoiled the type theory.

6. Discussion and Related Work

In this section, we consider some of the ramifications and consequences of the above result, as well as briefly consider some related work.

6.1. Linguistic observations

By Geuvers's Theorem, the derivation of induction we have given in $\iota\lambda P2$ is impossible in $\lambda P2$. Indeed it is important to emphasize that Geuvers proved that induction is uninhabited no matter what definition one gives for $Nat : \star$, $S : Nat \rightarrow Nat$, and $Z : Nat$. One can thus take the current result and Geuvers's together as showing that there is a true gap between $\iota\lambda P2$ and $\lambda P2$: $\iota\lambda P2$ cannot be reduced in a faithful way to $\lambda P2$. Let us consider this idea more closely. The first difference between $\iota\lambda P2$ and $\lambda P2$ is that $\iota\lambda P2$ is a type-assignment system (Curry-style), where $\lambda P2$ has annotated terms (Church-style – see [5] for more on the distinction). But Geuvers's result holds just as well for a type-assignment version of $\lambda P2$. This can be seen from the semantics for terms in Geuvers's model construction (Definition 7 of [16]), which ignores typing annotations. So the definition, and the soundness theorem based on it (Theorem 1 of [16]), works just as well for a type-assignment version of $\lambda P2$ as the annotated version.

So the essential difference between $\lambda P2$, where induction is not inhabited, and $\iota\lambda P2$, where it is, is in the implicit products, dependent intersections, and heterogeneous equality types. It is worth noting that the implicit products were necessary so that Church-encoded numbers could realize their own induction principles: both the number and the proof of its induction principle require, for the successor case, a function of one explicit argument, namely the result of the iteration/induction for the predecessor. If we were to use Parigot-encoded numbers, where each number contains its predecessor as a subterm, we could drop the implicit products and use instead positive-recursive types. Positive-recursive types (where the recursively defined type symbol can appear only positively in the body of the recursive type) are needed to type Parigot-encoded numbers anyway [28]. In this case, both the number and the proof of its induction principle would accept for the successor case a function of two (explicit) arguments, namely the predecessor number and the result of the iteration/induction for the predecessor. The development is very similar to the one above, so it is not presented here (though I have checked the code with the prototype implementation of $\iota\lambda P2$). For more on the Parigot encoding, see also [32].

6.2. Broader significance

This paper has shown a much simpler way than previous proposals to extend PTSs so that induction is derivable. We have seen this for natural-number induction, but it should be clear that the method applies to other inductive datatypes. Rather than add primitive inductive types to the core typed lambda calculus, we need only enrich our language of types to add enough dependent-typing power, so to speak, to the already existing computational power of a system like $\lambda P2$. As the base PTS $\lambda P2$ is a relatively tame PTS – many other PTSs will contain $\lambda P2$ as a sublanguage – the result in this paper is applicable to many other languages. For one obvious example, the Calculus of Constructions (CC) has $\lambda P2$ as a subsystem, and hence adding implicit products, dependent intersections, and heterogeneous equality to CC will make it possible to derive induction principles in that setting as well. One can envision an alternative history in which rather than change the underlying computational language of CC by adding primitive inductive types, we instead extend the language of types as proposed in this paper, and remain within a pure typed λ-calculus. So a system like Coq could have been founded instead on a PTS along the lines proposed here, rather than the Calculus of Inductive Constructions [24] (though see the conclusion for an important caveat regarding large eliminations).

What are the benefits of the proposed approach over primitive inductive datatypes? There are two: simplicity of the language, and expressive power. Even a casual inspection of works like Werner's dissertation should make it clear that defining a system of primitive inductive types is a complex matter. Because the type constructors themselves, and their term constructors, can have different arities (as well as there being different numbers of term constructors for different datatypes), the reduction and typing rules become quite heavy, with lots of vector notation to account for these differing arities. The addition of implicit products (and/or positive-recursive types), dependent intersections, and heterogeneous equality entails no such complexity, as we have seen above for $\iota\lambda P2$. This opens up the possibility of a much simpler approach to formalizing type theory within type theory, as has been the goal for a number of researchers for some time [2, 7, 9, 8]. It also should result in smaller trusted kernels for proof assistants based on type theory, as the rather heavy rules for primitive inductive types are not required.

For the second benefit over primitive inductive types: we have in this approach a potentially much more expressive language for datatypes than found in type theories with a system of primitive datatypes. The expressivity is along two dimensions. First, in Coq and the related language Agda, there are some strong restrictions placed on the form of datatypes [13]. In particular, both those systems require that each inductive type is mentioned only in strictly positive positions in the input types of its constructors. This rules out certain interesting idioms like higher-order encodings (see, e.g., [27]). With PTSs, the situation is different. If we wish to use Parigot encodings we have to accept the restriction to positive-recursive types. With Church encodings, however, there is absolutely no restriction: even datatypes with negative occurrences can be Church-encoded. Note, though, that exploring higher-order encodings in this setting is left to future work.

The second dimension of expressivity is in dependency. Type theorists have proposed inductive-recursive types and inductive-inductive types to allow greater intertwining between datatypes and either functions or other datatypes [3, 14]. With the present approach, such forms of datatypes should be definable more easily – though it remains to future work to confirm this. The reason is that here, we have a kind of two-level approach to datatypes. One first defines an impredicative type allowing only simple (non-dependent) eliminations, and then uses the dependent intersection to conjoin this type with the type of proofs of dependent eliminations over the first type. In the example above, we first defined the `cNat` type, and then the `Nat` type for which we could derive induction. This two-level approach should allow the second part of the definition, using the dependent intersection, to perform computations or reference constructors of other datatypes defined along with the first, non-dependent, part of the definition. So this should lead to an easy approach to inductive-inductive and inductive-recursive types, without any addition to the type theory. Indeed, Kopylov introduced dependent intersection types for similar reasons as Hickey's for introducing so-called very dependent function types: increased dependency, in their case for modeling dependent records in pure type theory [18].

6.3. Related work

The closest related work is my own paper under review at the time of this writing, on the Calculus of Dependent Lambda Eliminations (CDLE) [31]. The goal with CDLE is similar to that of the present paper: extend a PTS with new type forms for induction and dependently typed programming. CDLE adds a type construct called constructor-constrained recursive types. This is a form of recursive type based on the idea of preserving the typings of lambda-encoded constructors at each approximation (as familiar from fixed-point theory) to the recursive type. While adequate for deriving induction and while not requiring any change to the term language of the system (i.e., pure lambda calculus), constructor-constrained recursive types are still a rather complex feature, with fairly involved kinding rules. Their semantics requires a nontrivial extension to the already rather technical machinery needed for recursive types. The approach of this paper greatly improves on CDLE, by identifying a combination of reasonably simple typing constructs known already from the literature that suffice for deriving induction. No complex new typing construct is required. Indeed, except for the heterogeneous equality types, $\iota\lambda P2$ is a subsystem of CDLE. So just one simple addition is enough to obviate the entire complex machinery of constructor-constrained recursive types, which took several years to formulate and analyze. Needless to say, this was quite unexpected.

Several recent works have sought to find compact and powerful ways to add the complexity of a datatype system to a pure typed lambda calculus [1, 9]. Other works have sought deeper semantics for induction in type theory either categorically or through connections with parametricity [4, 21, 17]. We have already mentioned some of the historically decisive works which proposed adding primitive inductive types to pure type theory [33, 29, 11].

7. Conclusion

We have seen what I hope for the reader is a somewhat surprising result, namely that adding three constructs – implicit products, dependent intersections, and heterogeneous equalities – to the impredicative pure type system $\lambda P2$ is sufficient to derive induction. This overcomes Geuvers's Theorem on the underivability of induction in $\lambda P2$, by extending the language. We have confirmed that this extension has not trivialized the $\iota\lambda P2$ language as a logic, by giving a realizability semantics that implies logical consistency. We have also discussed some of the consequences of this result for devising simpler and at the same time more expressive constructive type theories.

Future work includes addressing one major omission in $\iota\lambda P2$, namely support for computing types from terms. This feature is required for full-fledged dependently typed programming, but was not relevant for our purpose in this paper. The CDLE language noted above includes a solution to the problem of large eliminations: a type construct for lifting terms to the type level, where they can then be used for type-level computation. We could add these lifting types to $\iota\lambda P2$ (or even better, a system ιCC extending the Calculus of Constructions with the three ingredients identified here for induction). A further point, since these type theories are closed (the syntax of types is not intended to be extended as the theory is developed), is to explore the inclusion of a universe. So there is more to do before the present approach can serve as a full-featured type theory. But the derivation of induction in a simple extension of an impredicative pure type system is a major step in the direction of a PTS suitable as a foundation for constructive type theory.

References

[1] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. PiSigma: Dependent Types without the Sugar. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming, 10th International Symposium (FLOPS)*, volume 6009 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2010.

[2] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 18–29. ACM, 2016.

[3] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A Categorical Semantics for Inductive-Inductive Definitions. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Algebra and Coalgebra in Computer Science - 4th International Conference, (CALCO)*, volume 6859 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2011.

[4] Robert Atkey, Neil Ghani, and Patricia Johann. A Relationally Parametric Model of Dependent Type Theory. *SIGPLAN Not.*, 49(1):503–515, January 2014.

[5] H. P. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[6] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types.* Perspectives in logic. Cambridge University Press, 2013.

[7] Bruno Barras. Sets in Coq, Coq in Sets. *J. Formalized Reasoning*, 3(1):29–48, 2010.

[8] James Chapman. Type Theory Should Eat Itself. *Electr. Notes Theor. Comput. Sci.*, 228:21–36, 2009.

[9] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming (ICFP)*, pages 3–14. ACM, 2010.

[10] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[11] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic*, pages 50–66, 1988.

[12] Haskell Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Science*, 20:584–590, 1934.

[13] The Agda development team. *Agda*, 2015. Version 2.4.2.2.

[14] Peter Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.*, 65(2):525–549, 2000.

[15] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The Expressiveness of Simple and Second-Order Type Structures. *J. ACM*, 30(1):151–185, 1983.

[16] Herman Geuvers. Induction Is Not Derivable in Second Order Dependent Type Theory. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications (TLCA)*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2001.

[17] Neil Ghani, Patricia Johann, and Clément Fumex. Fibrational Induction Rules for Initial Algebras. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop (CSL)*, volume 6247 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2010.

[18] Jason Hickey. Formal Objects in Type Theory Using Very Dependent Types. In *Foundations of Object-Oriented Languages (FOOL) 3*, 2003. Available from the NuPrl website, `www.nuprl.org`.

[19] William Howard. *The formulae-as-types notion of construction*, pages 479–491. Academic Press, 1980.

[20] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 86–95, 2003.

[21] Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL)*, volume 23 of *LIPIcs*, pages 432–451. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[22] Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 460–469. IEEE Computer Society, 1983.

[23] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[24] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2015. Version 8.4.

[25] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2002.

[26] Alexandre Miquel. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2001.

[27] Torben Æ. Mogensen. Efficient Self-Interpretations in lambda Calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[28] Michel Parigot. Programming with proofs: a second order type theory. In H. Ganzinger, editor, *European Symposium On Programming (ESOP)*, volume 300 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 1988.

[29] Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 5th International Conference*, pages 209–228, 1989.

[30] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[31] Aaron Stump. The Calculus of Dependent Lambda Eliminations, 2016. In second round of reviewing for Journal of Functional Programming.

[32] Aaron Stump and Peng Fu. Efficiency of lambda-encodings in total type theory. *Journal of Functional Programming*, 26, 003 2016.

[33] Benjamin Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris-Diderot - Paris VII, 1994.