Annals of Pure and Applied Logic 169 (2018) 637–655

Contents lists available at ScienceDirect

Annals of Pure and Applied Logic

www.elsevier.com/locate/apal

From realizability to induction via dependent intersection

Aaron Stump

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

ARTICLE INFO

Article history:
Received 16 October 2016
Received in revised form 8 January 2018
Accepted 7 March 2018
Available online 13 March 2018

MSC:
03B15
03B40
68N18
68N30

Keywords:
Extrinsic typing
Lambda encodings
Derivable induction
Internalized realizability

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 a built-in equality type. 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 [29,25,13]. Pure Type Systems (PTSs) are one formalism for constructive type theory, based on pure lambda calculus [6]. 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 [11]. 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 soft-

E-mail address: aaron-stump@uiowa.edu.

https://doi.org/10.1016/j.apal.2018.03.002
0168-0072/

ware [45,46,35,12]. 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 [20].

The adoption, in CIC and Coq, of a system for declaring primitive datatypes solved the problem of induction, and hence allowed formalization of a variety of results in Mathematics and Computer Science. A notable example among these is a Coq proof of the Four Color Theorem, which, unlike the theorem's original computer proof, does not depend on unverified programs for checking a large of number of special cases [23,4]. Thanks to the Curry–Howard isomorphism, such programs can be written and, crucially, proved sound within the type theory. So the addition of primitive datatypes opened up the possibility of formalizing complex mathematical results in type theory, that was lacking in pure CC. Other type theories provide mechanisms for defining inductive types. In Automath, for example, inductive types are defined axiomatically, simply by writing down constructors and asserting that induction holds [14]. In Martin-Löf type theory, one can use W-types to define inductive types, thus avoiding the need to add axioms to the theory for each new type; rather, the theory is extended once, with a single set of axioms for W-types [30]. Nevertheless, in all these cases, the pure type-theoretic core must be extended with additional operations, at both term and type level, to represent inductive types. At the term level, this necessitates additions to the usual proof of confluence of reduction of terms. In all cases, the resulting theory has now additional machinery requiring nontrivial metatheoretic analysis.

In this paper, we present an extension of a type-assignment formulation of $\lambda P2$, in which induction is derivable, indeed in two slightly different ways. 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 [32]. Second is the dependent intersection type of Kopylov [26]. Intersection types have been studied for many years in theoretical Computer Science, due to their strong connection with normalization properties (see [7] 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 primitive equality type, allowing expression of equality between terms x and y both of some common type A. While all three constructs are necessary for the derivations given of induction, the dependent intersections are most central to the construction, and so we will denote the resulting system $\iota\lambda P2$.

For nontrivial intersection types to be inhabited, we must work in a Curry-style (sometimes also called *extrinsic*) type theory, where we assign types to pure lambda terms. In such a theory, the same term can be assigned multiple inequivalent types. For example, assuming inequivalent types `Bool` and `Nat`, the term $\lambda\, x.\, x$ may be assigned the types $\texttt{Bool} \to \texttt{Bool}$ and $\texttt{Nat} \to \texttt{Nat}$. Church-style (also called *intrinsic*) type theories usually satisfy unicity of typing, by design: a given term has at most one type, modulo type equivalence. In the $\iota\lambda P2$ type theory we consider in this paper, the terms are only the terms of pure lambda calculus; i.e., variables, applications, and lambda abstractions. So we see that unlike the other approaches to inductive types mentioned above, the approach proposed here requires no additional constructs at the term level: terms remain just those of pure lambda calculus. We thus have a solution to the problem of induction in pure type theory (i.e., type theory whose terms are just the pure lambda-calculus terms). Of course, we must make some addition at the type level, or be blocked from deriving induction by Geuvers's result.

The centrality of dependent intersection for induction in $\iota\lambda P2$ is due to its role in internalizing a crucial realizability result of Leivant [28]. 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

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

Fig. 1. Syntax for $\iota\lambda P2$.

to the construction in this paper. We will use the dependent intersection type to define natural numbers to be those terms which are both Church-encoded natural numbers x (i.e., have type `cNat`, defined as usual to be $\forall$ `X : ⋆ . X → (X → X) → X`), and also proofs of induction, for predicates defined on `cNat`, for x. So `3`, for example, will be assigned both the type `cNat` and also `Inductive 3`, where `Inductive` is a predicate on `cNat x` stating that for all predicates `P` on `cNat`, if `P` holds of the `cNat 0` and is preserved by the usual successor operation on `cNat`s, then `P` holds of `x`. While dependent intersection types allow us to make this definition of `Nat`, it does not follow immediately that `Nat` is inductive. The reason is that the `Inductive` predicate expresses induction for `cNat`-predicates; it is not immediate that induction holds then for `Nat`-predicates. Nevertheless, we will see two ways, both somewhat subtle, to derive induction for `Nat`-predicates, given the definition of the type `Nat` as comprising intrinsically `cNat`-inductive `cNat`s.

Section 2 defines the $\iota\lambda P2$ type theory. This is a type-assignment system, and thus unsuitable for use as a type-checking algorithm. A system of annotations for terms must be devised to provide information that is otherwise missing when applying the type-assignment rules. In Section 3 we propose such a scheme, annotating terms with sufficient information to make typing essentially subject-directed. These annotations are inessential to the terms themselves, and are thus erased when checking convertibility of terms.

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. Indeed, two different inhabitants are constructed (Sections 4.4 and 4.5), in somewhat different ways. These constructions have 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. It also gives a suggestion of the more modest burden of basic metatheoretic analysis for the system. 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 Fig. 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 Fig. 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 [32]. $\iota\, x{:}T.\, T'$ 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 the equality type. Note that $\iota\lambda P2$, just like $\lambda P2$, does not allow the formation of type-level λ-abstractions over types. Fig. 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 (Fig. 2),
- $\Gamma \vdash T : \kappa$ expresses that type T has kind κ in context Γ (Fig. 3), and
- $\Gamma \vdash t : T$ expresses that term t may be assigned type T in context Γ (Fig. 4).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. The syntax for annotated terms.

The second and third rules in each figure are weakening rules. The last rule of Fig. 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 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. This is the bottom-left rule of Fig. 4; note that it arbitrarily uses $\lambda\, x.\, x$ as the proof of a trivial equality.

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 bidirectional type checking [36]. 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 Fig. 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

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

Fig. 6. Eraser function for annotated terms.

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

Fig. 7. Modified kinding rules referring to annotated terms.

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

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

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

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

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

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

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

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

Fig. 8. Bidirectional typing rules for annotated terms.

should now be understood to be to annotated terms. Annotated terms erase to unannotated ones as shown in Fig. 6. We extend this function to types, kinds, and contexts in the obvious way, by applying the eraser function of Fig. 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 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 Fig. 7.

The typing rules for annotated terms are in Fig. 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. There is one caveat to this: occasionally it is necessary to change a type to a β-equivalent one, in order to expose an opportunity for equality elimination (ρ). For this, my $\iota\lambda P2$ implementation provides an explicit annotation

to allow the user of the system to convert a type to the exact form required to do a rewrite. We write $\chi\ T\ -\ t$ for an explicit conversion, with the following typing rule:

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

The erasure of $\chi\ T\ -\ t$ is just the erasure of t.

The rules of Fig. 8 are in one–one correspondence with the rules of Fig. 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 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, and both have some common type T. 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* $\iota\lambda P2$*).*

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'Don-

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

Fig. 9. Definition of Church-encoded natural numbers and their constructors.

```
Inductive ◁ cNat → ⋆ = λ x : cNat .
  ∀ P : cNat → ⋆ . P cZ → (∀ y : cNat . P y → P (cS y)) → P x.
iZ ◁ Inductive cZ = Λ X . λ z . λ s . z .
iS ◁ ∀ x : cNat . Inductive x → Inductive (cS x) =
 Λ x . λ p . Λ P . λ z . λ s . s -x (p · P z s) .
```

Fig. 10. The `Inductive` predicate and its constructors.

```
Nat ◁ ⋆ = ι x : cNat . Inductive x.
Z ◁ Nat = [ cZ , iZ ] .
S ◁ Nat → Nat = λ n . [ cS n.1 , iS -n.1 n.2 ] .
```

Fig. 11. Definition of `Nat` type.

nell [19]). This is done in Fig. 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 predicate `Inductive`

Next, we define a predicate `Inductive` on `cNat`, expressing that a Church-encoded natural number `x` is inductive: for any predicate `P` on `cNat`, if `P` holds of `cZ` and is preserved by `cS`, then it holds of `x`. We are using an implicit product type in the statement of the successor (step) case of induction, namely `∀ x : cNat . P x → P (cS x)`. This is critical for internalizing Leivant's observation, as we can see in the definitions of constructors `iZ` and `iS` (also Fig. 10) for the `Inductive` predicate. Another way of phrasing Leivant's observation is to say that the constructors `cZ` and `iZ` have the same erasure – as indeed they are easily seen to have – and so too `cS` and `iS`. It is for the latter that the use of implicit products is crucial, for it ensures that the body `s -x (p · P z s)` of `iS` erases to `s (p z s)`, which is indeed the erasure of the body of `cS`. If instead we had a Π-abstraction `Π x : cNat . P x → P (cS x)` for the statement of the step case, the body of `iS` would be `s x (p · P z s)`, whose erasure is `s x (p z s)`; this would not match the erasure of the body of `cS`. So for the definitions of the constructors of `cNat` and `Inductive` to **align**, we need to use implicit products in the definition of `Inductive`.

4.3. The type `Nat`

We may now define the type `Nat`, in Fig. 11. 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 Fig. 11) is critical here, to allow us to express that x realizes its own induction principle. The constructors `Z` and `S` are then defined (also Fig. 11) using the annotated term construct `[t,t']` to introduce dependent intersections. Since `n.1` and `n.2` both erase to `n`, and since we already observed that the constructors `cZ` and `iZ`, and `cS` and `iS` have the same erasures, we see that the two components of the dependent-intersection introduction have the same erasure in each case.

```
Induction ◁ Π n : Nat . ∀ P : Nat → ⋆ . P Z → (∀ m : Nat . P m → P (S m)) → P n =
  λ n . Λ P . λ z . λ s .
    n.2 · (λ x : cNat . ∀ X : ⋆ . (Π m : Nat . (x ≃ m.1) → P m → X) → X )
     (Λ X . λ c . c Z β z)
     (Λ x . λ ih . Λ X . λ c .
       ih · X (λ m . λ e . λ u . c (S m) (ρ e - β) (s -m u)))
     · (P n) (λ m . λ e . λ u . ρ e - u).
```

Fig. 12. Derivation of induction, first method.

4.4. Proving induction for `Nat`*, first method*

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

```
Π n : Nat . ∀ P : Nat → ⋆ . P Z → (∀ m : Nat . P m → P (S m)) → P n
```

Informally, here is the basic idea. We take in arguments `n`, `P`, `z`, and `s`, for the first four abstractions in that type. We will then use `n.2` to prove the following predicate on `n.1` (of type `cNat`). The predicate holds of `x` of type `cNat` just in case there exists an `m` of type `Nat`, such that `m.1` equals `x` and `P m` holds. So our use of `n.2` will actually compute a triple (`m`, proof of equality, proof of `P m`). This triple needs to be Church-encoded, since $\iota\lambda P2$ does not provide tuples (or any other datatype!) natively.

Constructing this triple 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 `m` of type `Nat`, and the latter builds the proof of `P m`. 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 `n.2`, then we will get a Church-encoded number also representing `n.2`. The equality type allows us to internalize this observation, stating that `m.1` (where m is the `Nat` we are constructing) is equal to the $n.1$ for which $n.2$ is allowing us to perform a dependent elimination. Since the `.1` annotations disappear in erasure, we can recover a proof of `P n` at the end of the iteration.

Let us now work through the details of the derivation in $\iota\lambda P2$, shown in Fig. 12. To aid the patient reader, I am using variable `x` to range over `cNat`, and `n` and `m` to range over `Nat`. We are defining `Ind` whose type is the induction principle for `Nat`. The derivation of this principle begins after the equals sign, with `ΛP`. We first take the following inputs:

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

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

```
∀ P : cNat → ⋆ . P cZ → (∀ y : cNat . P y → P (cS y)) → P n.1
```

So the first thing we must do when performing an elimination with `n.2` is to supply the instance of `P` (what is sometimes called the **motive** [31]), which is the predicate discussed earlier; in the code of Fig. 12 it is on the third line from the top of the code:

```
λ x : cNat . ∀ X : ⋆ . (Π m : Nat . (x ≃ m.1) → P m → 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 `n.1`, by dependent iteration:

```
∀ X : ⋆ . (Π m : Nat . (n.1 ≃ m.1) → P m → X) → X
```

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

- `m` of type `Nat`,
- a proof of `n.1 ≃ m.1`, and
- a proof of `P m`

Once we have computed this triple, we can extract a proof of `P n`, as done in the bottom line of Fig. 12: we instantiate the type variable `X` in the type R with `P n`, and then return the third component of the triple, casting `P m` to `P n` using the second component. This is possible since `n.1 ≃ m.1` erases to `n ≃ m`.

We must look now at the zero and successor cases of the dependent elimination of `n.2`, to complete our detailed examination of the code of Fig. 12. The zero case is first, on the fourth line from the top of Fig. 12:

```
Λ X . λ c . c Z β z
```

We take in inputs `X` of kind $\star$ and `c` of type

```
Π m : Nat . ((cZ ≃ m.1) → (P m) → 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 `P 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 m . λ e . λ u . c (S m) (ρ e - β ) (s -m u))
```

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

- `x` of type `cNat`
- `ih` of type `∀ X : ⋆ . ((Π m : Nat . ((x ≃ m.1) → (P m) → X)) → X)`
- `X` of kind $\star$
- `c` of type `Π m : Nat . (((cS x) ≃ m.1) → (P m) → 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 Fig. 12):

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

```
toNat ◁ cNat ⟶ Nat = λ x . x · Nat Z S .

reflection ◁ Π n : Nat . toNat n.1 ≃ n =
  λ n . n.2 · (λ x : cNat . (toNat x).1 ≃ x)
          β
          (Λ x . λ ih . χ (cS ((toNat x).1) ≃ cS x) - ρ ih - β) .
```

Fig. 13. Converting `cNat` to `Nat` is extensionally the identity.

- `m` of type `Nat`,
- `e` of type `x ≃ m.1`, and
- `u` of type `P m`.

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

- `S m` of type `Nat`,
- ρ `e -` β of type `(cS x) ≃ (S m).1`, and
- `s -m u` of type `P (S m)`.

Let us look carefully at the typings for each of these components. First, since `m` is of type `Nat`, we have `S m` also of type `Nat`, since `S` is of type `Nat → Nat`. Next, to prove `(cS x) ≃ (S m).1`, it suffices to rewrite `x` to `m` and then check that `cS m` is β-equivalent to `S m`. 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 `m`. We must also supply a proof of `P m`, and this is `u`. This completes the detailed examination of the code in Fig. 12.

4.5. Proving induction for `Nat`, second method

Given the definitions of `cNat` and `Nat` above (Figs. 9, 10, and 11), there is an alternative way to derive induction, which we consider now. The first step is to define a function `toNat` that converts a `cNat` (call it `x`) to a `Nat`. This is easily done just by applying `x` to the constructors for `Nat` (i.e., `Z` and `S`), as shown in Fig. 13. The figure then proves a theorem, under the standard name `reflection`, which says that `toNat` is extensionally the identity: applying a `Nat n` to the `Nat` constructors just has the effect of rebuilding the number. This theorem follows directly from universality of the following predicate on `cNat x`: `(toNat x).1 ≃ x`. The theorem follows from this predicate applied to `n.1`.

The proof in Fig. 13 uses the form of induction provided by `n.2`, which is applicable because the predicate just mentioned is a predicate on `cNat` (not `Nat`). The proof of the base case (in Fig. 13) is just β. The step case is a little more interesting:

```
Λ x . λ ih . χ (S (toNat x) ≃ S x) - ρ ih - β
```

We are given `x` of type `cNat` and `ih` of type `toNat x ≃ x`. We must prove

```
(toNat (cS x)).1 ≃ cS x
```

We use an explicit conversion (with χ) to change the goal type to the definitionally equivalent equation `cS ((toNat x).1) ≃ cS x`. Note that the sides of this equation still type-check, at type `cNat`. The crucial point of this use of χ is to expose the subterm `toNat x`, which may then be rewritten using `ih` to just `x`. This renders the goal trivial, completing the proof.

The next step is the derivation of induction, shown in Fig. 14. This derivation is simpler than the one we saw in the previous section. It takes in `n` of type `Nat`, and then the predicate `P` and base and step cases `z` and `s`. Then using `n.2`, it shows that the predicate `λ x : cNat . P (toNat x)` is universal; that is, that

```
symm ◁ ∀ T : ★ . ∀ t : T . ∀ t' : T . (t ≃ t') → t' ≃ t
  = Λ T . Λ t . Λ t' . λ u . ρ u - β.

Induction2 ◁ Π n : Nat . ∀ P : Nat → ★ . P Z → (∀ m : Nat . P m → P (S m)) → P n =
  λ n . Λ P . λ z . λ s .
    ρ (symm · Nat -(toNat n.1) -n (reflection n)) -
      (n.2 · (λ x : cNat . P (toNat x))
        z (Λ p . λ q . s -(toNat p) q)) .
```

Fig. 14. Derivation of induction, second method.

```
compInduction2
  ◁ ∀ n : Nat . ∀ P : Nat → ★ . ∀ z : P Z . ∀ s : ∀ m : Nat . P m → P (S m) .
      Induction2 (S n) z s ≃ s (Induction2 n z s)
= Λ n . Λ P . Λ z . Λ s . β .
```

Fig. 15. Deriving the expected reduction for incomplete values `S n`.

`P` holds of the conversion of a `cNat x` to a `Nat`. The `reflection` theorem (of Fig. 13) then lets us drop this call to `toNat`. So the body of the derivation (of Fig. 14) indeed has type `P n`. An easily derived lemma of symmetry of equality (`symm` in Fig. 14) is used so that we change `n` in the goal type to `toNat n.1`, before checking the term headed by `n.2`, since that term proves `P (toNat n.1)`.

A remarkable point: the erasure of `Induction2` is `λ n . λ z . λ s . n z (λ q . s q)`. From this term we have the following chain of η-equivalences:

$$\begin{aligned}
&\lambda n.\lambda z.\lambda s.n\ z\ (\lambda q.s\ q) &=_\eta\\
&\lambda n.\lambda z.\lambda s.n\ z\ s &=_\eta\\
&\lambda n.\lambda z.n\ z &=_\eta\\
&\lambda n.n
\end{aligned}$$

Here we see a very noteworthy difference between `Induction2` and `Induction`. The latter is not η-equivalent to the identity function, due to the need to destruct and construct triples during the iteration. If we added native existential types to the theory, then likely a version of `Induction` using such types would be η-equivalent to the identity.

The fact that `Induction2` is extensionally equivalent to the identity function means that not only does it derive the desired reasoning principle, but also its reduction behavior is what one would like to have for computational use. In particular, we can prove the usual reduction rule for using `Induction2` as an iterator with an incomplete value. This is shown in Fig. 15. Given a `Nat n`, predicate `P`, base and step cases `z` and `s`, we see that using `Induction2` as an iterator indeed allows us to permute it over `S`. Furthermore, this permutation follows solely by β-reduction; hence the β in the last line of Fig. 15, for proving the equation expressing the desired reduction behavior. This says that `Induction2` behaves computationally like an iterator. This facilitates external reasoning about functions defined using `Induction2`.

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

In this section, we sketch a realizability semantics for $\iota\lambda P2$ types, which may be used to show logical consistency of $\iota\lambda P2$. We use a simplified form of a similar semantics proposed in previous work [39]. 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.

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

Fig. 16. Semantics for types and kinds.

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

Fig. 17. Semantics of typing contexts Γ.

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. This is just a mechanism to obtain a representative of E, which in effect allows us to use E as a term (for example, in instantiating the body of $\lambda\, x.\, t$ in the clause defining the semantics of Π-types, in Fig. 16).

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.*

Fig. 16 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 $\cap_\star$, defined at the end of Fig. 16, 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.

Critically, the semantics makes use of Girard's technique for interpreting impredicative higher-order quantification: we interpret a syntactic quantification over types via a semantic quantification over reducibility candidates. This is in the clause defining $[\![\forall X : \kappa.T]\!]_{\sigma,\rho}$ in Fig. 16, where the intersection over all $S \in [\![\kappa]\!]_{\sigma,\rho}$ is essentially expressing a universal quantification over all such S; that is, a term t is in the interpretation of $\forall X : \kappa.T$ iff it is in $[\![T]\!]_{\sigma,\rho[X\mapsto S]}$ for all $S \in [\![\kappa]\!]_{\sigma,\rho}$. So we interpret syntactic impredicative quantification by means of semantic impredicative quantification.

Fig. 17 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. □

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 equality types to $\lambda P2$ has not spoiled the type theory. While it is customary to prove several other properties about a type theory such as $\iota\lambda P2$ – normalization (weak or strong) and type preservation, chiefly – those are not undertaken here. Instead, I have opted to present the simplest realizability semantics for types I know how to give for this system, which provides what I hope is a clear definition of what the types are truly intended to mean; and with respect to which the typing rules are sound.

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 $\texttt{Nat} : \star$, $S : \texttt{Nat} \to \texttt{Nat}$, and $Z : \texttt{Nat}$. 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 [6] 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 [20]), which ignores typing annotations. So the definition, and the soundness theorem based on it (Theorem 1 of [20]), 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 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 [34]. 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 [40].

6.2. Broader significance

This paper has shown a much simpler way than previous proposals to extend PTSs so that induction is derivable. 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 homogeneous equality types 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 [45] (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 equality types 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 [2,8,10,9]. 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 [44]. 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., [33]). 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 [3,16]. 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 could lead to a new 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 [24].

6.3. Treating classes of inductive definitions

The form of induction we have derived in this paper is standard natural-number induction. It is reasonable to ask, what classes of inductive definitions can be supported in a similar manner? Following the initial submission of this paper, Denis Firsov and I have taken some steps to answering this question. In particular, we have shown how to define the least fixed-point of an arbitrary functor F, together with an induction principle for F, in the natural extension of $\iota\lambda P2$ with type-level functions (needed even to be able to express the idea of a functor F mapping types to types). So we derive induction for any inductive type that can be expressed by a type scheme F of kind $\star \to \star$, a term *fmap* of type

$$\forall X{:}\star.\, \forall Y{:}\star.\, (X \to Y) \to F\ X \to F\ Y$$

together with terms whose types express the identity and composition laws for F [18]. We have found (but not published) that it is straightforward to make that construction work for indexed functors, from which one can then derive mutually inductive types, as well as indexed types like the standard example of vectors.

6.4. Metamathematical perspectives: complexity

As sketched above, consistency of $\iota\lambda P2$ is relatively easy to establish, while consistency proofs for CIC or Martin-Löf Type Theory (MLTT) with various features are generally more involved. Now in some cases, the complexity ensues from higher goals for the metatheoretic analysis. For example, ordinal analysis of a version of MLTT as in [38], or categorical semantics as in, for example, [15]. These are more complex endeavors than simply giving a single quotiented-term model, in order to prove logical consistency (as sketched in Section 5). But in other cases, the additional complexity really does arise from the additional technical details required to formulate the theory. For example, the analysis of CIC, notably more complex than that of $\iota\lambda P2$, is based on a similar realizability semantics – coming, as ours here, from Girard [22] – though aimed at more than just logical consistency, namely strong normalization [46]. There is more metatheoretic work to be done, as noted already, when the term language of the underlying pure type theory must be extended with constants for constructors of datatypes and eliminators. At the very least, the confluence proof of the underlying lambda calculus must be extended.

By deriving natural-number induction in $\iota\lambda P2$, we have a relatively simple constructive foundation for the fundamental mathematical concept of natural number. A point well worth emphasizing is that the relative simplicity of $\iota\lambda P2$ is coming from the power of impredicative type theory. Girard's analysis of impredicative quantification, while not contributing to goals like ordinal analysis of various higher-order theories (indeed, ordinal analysis of full second-order arithmetic is still to be achieved), provides a simple technique for establishing consistency of the theory (relative, of course, to the consistency of the background metatheory).

But where other theories like CIC or MLTT incur additional complexity for inductive datatypes, impredicative quantification, which is from a proof-theoretic perspective much more powerful, results in a simpler treatment. This seems like a benefit of the approach here.

One could argue against this point, in saying that the full power of impredicativity is not needed to develop a suitable foundation for mathematics. And certainly if one's goal were to devise the proof-theoretically weakest foundation suitable for mathematical reasoning, the approach proposed here would be a spectacular loser. Feferman has argued, for example, that with a few exotic exceptions, all of scientifically applicable mathematics can be formulated in his theory W (for Weyl), which is a conservative extension of Peano Arithmetic (and hence vastly weaker in proof-theoretic strength than an impredicative type theory like $\iota\lambda P2$) [17].

On the other hand, if one's goal is to develop a powerful constructive type theory based on a core of minimal formal size (if not minimal proof-theoretic strength), than we have seen evidence that the approach proposed here is currently the best available: a compact pure type theory, just CC plus three additional typing constructs, in which natural-number induction can be derived. No other work I am aware of matches this result for minimality of the core theory.

A final note on connections with proof theory: Pimentel et al. have carried out a very interesting proof-theoretic study of intersection types, seeking to show, among other things, how intersection types can indeed be viewed as a special form of conjunction, governed by different proof rules than the usual conjunction [37]. This gives a logical analysis of intersection types. It would be interesting to see if the authors' analysis could be extended to the dependent intersections used in this paper. The authors also raise the interesting question of which implicational operator would play the adjunctive role for the intersection type which the usual intuitionistic implication plays for the standard conjunction. In $\iota\lambda P2$, I anticipate using Church-encoded pairs for standard conjunction, though certainly it is only reasons of minimality that would prevent one from combining, as Pimentel et al. do, intersections and conjunctions in one theory.

6.5. Metamathematical perspectives: logicism

Another metamathematical question one may ask regarding the results of this paper is, what light, if any, they shed on the question(s) of logicism. Full consideration of this point is beyond the scope of the present paper, but I would like to offer a few remarks on this. First, let us take the following proposition from Tennant's Stanford Encyclopedia of Philosophy entry on logicism as a fair rough description of the basic doctrine [43]:

> "Logic is capable of furnishing definitions of the primitive concepts of these branches of mathematics [arithmetic and real analysis], allowing one to derive the mathematician's 'first principles' therein as results within Logic itself."

If one counts $\iota\lambda P2$ as a logic, then certainly at least as regards arithmetic, the derivations in $\iota\lambda P2$ seem to support logicism as just formulated. For with no further axioms or extensions, we have defined natural numbers and proven the induction principle for them, in $\iota\lambda P2$.

We must ask, however, if $\iota\lambda P2$ can indeed be viewed as a logic, not adulterated with some additional nonlogical principles. A related question is, to the existence of which entities does $\iota\lambda P2$ seem to be committed? Here, the simple answer which suggests itself to me is, that $\iota\lambda P2$ is committed to the existence, in some form, of the terms, or some semantic objects those terms denote, of pure untyped lambda calculus. Does one wish to maintain that such terms should be viewed as logical entities (of whatever metaphysical status)? Here I wish to leave the discussion, as debating the exact connection, with any metaphysical consequences, between constructive logic and programs seem likely to involve both intricacy and controversy.

6.6. Related work

The closest related work is my own paper on the Calculus of Dependent Lambda Eliminations (CDLE) [39]. 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 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.

Another point of comparison between CDLE and $\iota\lambda P2$ is that CDLE includes a lifting operator, that translates simply-typed terms into simply-kinded types. This allows one to implement so-called *large eliminations*, where types may be computed by recursion on (in this case, lambda-encoded) data. Lifting is omitted from $\iota\lambda P2$ for simplicity, but its absence does mean that we cannot prove negative facts like $0 \neq 1$ expressing disjointness of the ranges of constructors: the standard proofs of these, even with built-in inductive types, rely on large eliminations. For a full-fledged type theory, one would indeed want to add to $\iota\lambda P2$ lifting or some similar mechanism, to allow derivation of such facts. And indeed, one would also like to have type-level functions, which CDLE includes (following CC and F_ω), but $\iota\lambda P2$ has excluded, in order to keep the type theory as small as possible for deriving natural-number induction.

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 [1,10]. Other works have sought deeper semantics for induction in type theory either categorically or through connections with parametricity [5,27,21]. We have already mentioned some of the historically decisive works which proposed adding primitive inductive types to pure type theory [46,35,12].

A related neo-logicist effort is in [42], where Tennant derives natural-number induction based on a definition of a natural-number predicate in terms of a relation expressing that a value r can be reached from a value t by successive applications of a function f. That relation is introduced using meta-level predicate quantification to express abstractly the condition about successive applications of f. Tennant's development takes place within an intuitionistic relevant logic. The use of meta-level quantification allows him to avoid committing to a second-order logic, but has the consequence of forcing him to add new predicate symbols and terms to the language, with corresponding introduction and elimination rules, instead of making explicit definitions in terms of second-order quantifications. This move to make use of meta-level predicate quantification saves Tennant from committing to a second-order language, at the cost of requiring new metatheoretic analysis to justify soundness of each concept he adds (Tennant does not undertake such an analysis in [42]). In contrast, in $\iota\lambda P2$, the power of explicit second-order quantification within the language allows us to analyze the theory once and for all, and then introduce new terms via explicit definition, rather than as new constructs with new logical rules. A further important difference is that where Tennant works within a logic, $\iota\lambda P2$ is a type theory, and thus supports not just formal reasoning, but also dependently typed programming, which is of notable current interest in Computer Science [41].

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 equality types – 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 adding the lifting types (mentioned in Section 6.6 above) to $\iota\lambda P2$. Going further, $\iota\lambda P2$ should be extended to a system ιCC based on the extrinsic Calculus of Constructions, extended 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.

Acknowledgements

Many thanks to the anonymous APAL referee for very thoughtful and constructive discussion of an earlier draft of the paper, and to the editor Martin Hyland for his support during the revision. Thanks also to Denis Firsov for observing that a homogeneous equality type is sufficient – an earlier draft used heterogeneous equality (thanks also to the referee for noting this, too). Thanks to Larry Diehl for observing that the first derivation of induction (Fig. 12) does not exhibit the desired reduction behavior on incomplete values. I also gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program). Thanks to Madeliene, Seraphina, and Oliver. AMDG.

References

[1] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, Nicolas Oury, PiSigma: dependent types without the sugar, in: Matthias Blume, Naoki Kobayashi, Germán Vidal (Eds.), Functional and Logic Programming, 10th International Symposium (FLOPS), in: Lecture Notes in Computer Science, vol. 6009, Springer, 2010, pp. 40–55.
[2] Thorsten Altenkirch, Ambrus Kaposi, Type theory in type theory using quotient inductive types, in: Rastislav Bodík, Rupak Majumdar (Eds.), Proceedings of the 43rd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), ACM, 2016, pp. 18–29.
[3] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, Anton Setzer, A categorical semantics for inductive–inductive definitions, in: Andrea Corradini, Bartek Klin, Corina Cîrstea (Eds.), Algebra and Coalgebra in Computer Science – 4th International Conference (CALCO), in: Lecture Notes in Computer Science, vol. 6859, Springer, 2011, pp. 70–84.
[4] K. Appel, W. Haken, Every planar map is four colorable, Bull. Amer. Math. Soc. 82 (5) (1976) 711–712.
[5] Robert Atkey, Neil Ghani, Patricia Johann, A relationally parametric model of dependent type theory, SIGPLAN Not. 49 (1) (January 2014) 503–515.
[6] H.P. Barendregt, Lambda calculi with types, in: S. Abramsky, Dov M. Gabbay, S.E. Maibaum (Eds.), Handbook of Logic in Computer Science (vol. 2), Oxford University Press, Inc., New York, NY, USA, 1992, pp. 117–309.
[7] Hendrik Pieter Barendregt, Wil Dekkers, Richard Statman, Lambda Calculus with Types, Perspectives in Logic, Cambridge University Press, 2013.
[8] Bruno Barras, Sets in Coq, Coq in sets, J. Formaliz. Reason. 3 (1) (2010) 29–48.
[9] James Chapman, Type theory should eat itself, Electron. Notes Theor. Comput. Sci. 228 (2009) 21–36.
[10] James Chapman, Pierre-Évariste Dagand, Conor McBride, Peter Morris, The gentle art of levitation, in: Paul Hudak, Stephanie Weirich (Eds.), Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP), ACM, 2010, pp. 3–14.
[11] T. Coquand, G. Huet, The calculus of constructions, Inform. and Comput. 76 (2–3) (1988) 95–120.
[12] Thierry Coquand, Christine Paulin, Inductively defined types, in: Per Martin-Löf, Grigori Mints (Eds.), COLOG-88, International Conference on Computer Logic, 1988, pp. 50–66.
[13] Haskell Curry, Functionality in combinatory logic, Proc. Natl. Acad. Sci. 20 (1934) 584–590.

[14] N.G. de Bruijn, AUTOMATH, a Language for Mathematics, Technical Report T.H. Report 66-WSK-05, Department of Mathematics, Eindhoven University of Technology, 1968.
[15] Peter Dybjer, Internal type theory, in: Stefano Berardi, Mario Coppo (Eds.), Types for Proofs and Programs, Selected Papers, International Workshop TYPES'95, Torino, Italy, June 5–8, 1995, in: Lecture Notes in Computer Science, vol. 1158, Springer, 1996, pp. 120–134.
[16] Peter Dybjer, A general formulation of simultaneous inductive–recursive definitions in type theory, J. Symbolic Logic 65 (2) (2000) 525–549.
[17] Solomon Feferman, In the Light of Logic, 1998.
[18] Denis Firsov, Aaron Stump, Generic derivation of induction for impredicative encodings in Cedille, in: June Andronick, Amy Felty (Eds.), Certified Programs and Proofs (CPP), 2018.
[19] Steven Fortune, Daniel Leivant, Michael O'Donnell, The expressiveness of simple and second-order type structures, J. ACM 30 (1) (1983) 151–185.
[20] Herman Geuvers, Induction is not derivable in second order dependent type theory, in: Samson Abramsky (Ed.), Typed Lambda Calculi and Applications (TLCA), in: Lecture Notes in Computer Science, vol. 2044, Springer, 2001, pp. 166–181.
[21] Neil Ghani, Patricia Johann, Clément Fumex, Fibrational induction rules for initial algebras, in: Anuj Dawar, Helmut Veith (Eds.), Computer Science Logic, 24th International Workshop (CSL), in: Lecture Notes in Computer Science, vol. 6247, Springer, 2010, pp. 336–350.
[22] Jean-Yves Girard, Paul Taylor, Yves Lafont, Proofs and Types, Cambridge University Press, New York, NY, USA, 1989.
[23] Georges Gonthier, Formal proof – the four-color theorem, Notices Amer. Math. Soc. 55 (11) (2008) 1382–1393.
[24] 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.
[25] William Howard, The Formulae-as-Types Notion of Construction, Academic Press, 1980, pp. 479–491.
[26] Alexei Kopylov, Dependent intersection: a new way of defining records in type theory, in: 18th IEEE Symposium on Logic in Computer Science (LICS), 2003, pp. 86–95.
[27] Neelakantan R. Krishnaswami, Derek Dreyer, Internalizing relational parametricity in the extensional calculus of constructions, in: Simona Ronchi Della Rocca (Ed.), Computer Science Logic 2013 (CSL), in: LIPIcs, vol. 23, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013, pp. 432–451.
[28] Daniel Leivant, Reasoning about functional programs and complexity classes associated with type disciplines, in: 24th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 1983, pp. 460–469.
[29] Per Martin-Löf, Intuitionistic Type Theory, Bibliopolis, Napoli, 1984.
[30] Per Martin-Löf, Intuitionistic type theory, in: Notes by Giovanni Sambin of a Series of Lectures Given in Padua, June 1980, Bibliopolis, 1984.
[31] Conor McBride, Elimination with a motive, in: Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack (Eds.), Types for Proofs and Programs, Selected Papers, International Workshop, TYPES 2000, Durham, UK, December 8–12, 2000, in: Lecture Notes in Computer Science, vol. 2277, Springer, 2002, pp. 197–216.
[32] Alexandre Miquel, The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping, in: Samson Abramsky (Ed.), Typed Lambda Calculi and Applications, in: Lecture Notes in Computer Science, vol. 2044, Springer, 2001, pp. 344–359.
[33] Torben Æ. Mogensen, Efficient self-interpretations in lambda calculus, J. Funct. Programming 2 (3) (1992) 345–363.
[34] Michel Parigot, Programming with proofs: a second order type theory, in: H. Ganzinger (Ed.), European Symposium on Programming (ESOP), in: Lecture Notes in Computer Science, vol. 300, Springer, 1988, pp. 145–159.
[35] Frank Pfenning, Christine Paulin-Mohring, Inductively defined types in the calculus of constructions, in: Michael G. Main, Austin Melton, Michael W. Mislove, David A. Schmidt (Eds.), Mathematical Foundations of Programming Semantics, 5th International Conference, 1989, pp. 209–228.
[36] Benjamin C. Pierce, David N. Turner, Local type inference, ACM Trans. Program. Lang. Syst. 22 (1) (2000) 1–44.
[37] Elaine Pimentel, Simona Ronchi Della Rocca, Luca Roversi, Intersection types from a proof-theoretic perspective, Fund. Inform. 121 (1–4) (2012) 253–274.
[38] Anton Setzer, Well-ordering, proofs for Martin-Löf type theory, Ann. Pure Appl. Logic 92 (2) (1998) 113–159.
[39] Aaron Stump, The calculus of dependent lambda eliminations, J. Funct. Programming 27 (2017) e14.
[40] Aaron Stump, Peng Fu, Efficiency of lambda-encodings in total type theory, J. Funct. Programming 26 (2016) 003.
[41] Wouter Swierstra, Peter Dybjer, Special issue on Programming with Dependent Types Editorial, J. Funct. Programming 27 (2017) e15.
[42] Neil Tennant, Deriving basic laws of arithmetic, in: Anti-Realism and Logic: Truth as Eternal, Oxford University Press, 1987, chapter 25.
[43] Neil Tennant, Logicism and neologicism, in: Edward N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy, winter 2017 edition, Metaphysics Research Lab, Stanford University, 2017.
[44] The Agda development team, Agda, 2015. Version 2.4.2.2.
[45] The Coq development team. The Coq proof assistant reference manual. LogiCal Project, 2015. Version 8.4.
[46] Benjamin Werner, Une Théorie des Constructions Inductives, PhD thesis, Université Paris-Diderot – Paris VII, 1994.