Type Theories from Barendregt's Cube for Theorem Provers*

Jonathan P. Seldin
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
jonathan.seldin@uleth.ca
http://home.uleth.ca/∼jonathan.seldin

November 28, 2001

Abstract

Anybody using a theorem prover or proof assistant will want to have confidence that the system involved will not permit the derivation of false results. On some occasions, there is more than usual need for this confidence. This paper will discuss some logical systems based on typed lambda-calculus that can be used for this purpose. The systems are natural deduction systems, and use the propositions-as-types paradigm. Not only are the underlying systems provably consistent, but additional unproved assumptions from which a lot of ordinary mathematics can be derived can also be proved consistent. Finally, the systems have few primitive postulates that need to be programmed separately, so that it is easier for a programmer to see whether the code really does program the systems involved without errors.

Theorem provers and proof assistants are used for a variety of purposes. For some of these purposes, including many cases of formal verification, these theorem provers must be *trusted*. Among the conditions needed for a trusted system are the following:

*This work was supported in part by grant RGP-23391-98 from the Natural Sciences and Engineering Research Council of Canada.

- *Consistency*. It must not be possible to derive a contradiction in the system.
- *Confidence in the implementation*. The user must have good reason to believe that the coding for the implementation really does accurately program the formal logic involved, and does not accidently include extra principles.

In some cases, it is also necessary for the theorem prover to be small, for example in *proof carrying code* (PCC) [1, 2]. PCC is designed to provide computer users installing new software evidence that the software is safe: the software code is to include a formal proof that the software satisfies certain safety conditions (such as not writing to the wrong memory locations), and the computer on which the software is installed is to have a theorem prover that checks this formal proof. In order not to interfere with other computer operations, the theorem prover needs to be small. In order to provide the necessary assurance to the user, it must be trusted.

In this paper, we will look at some formal systems that can be used for theorem provers or proof assistants of this kind.

I would like to thank Roger Hindley for his helpful comments and suggestions.

1 Barendregt's λ-cube

Definition 1 The λ-cube of Barendregt [3] is a collection of eight systems of type assignment to λ-calculus. The systems all have the same syntax:

$$M \longrightarrow x|c|\mathsf{Prop}|\mathsf{Type}|(MM)|(\lambda x : M\ .\ M)|(\forall x : M)M.$$

Here Prop and Type are special constants called *sorts*; s, s', and s_1, etc., will be used for sorts.[1] Conversion will be β-conversion, generated by

$$(\lambda x : A\ .\ M)N \rhd [N/x]M,$$

where $[N/x]M$ denotes the substitution of N for all free occurrences of x in M, with bound variables being changed to avoid conflicts. Judgements are of the form $\Gamma\ \vdash\ M : A$, where Γ is

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n.$$

[1] It is common to use * for Prop, $\Box$ for Type. I formerly referred to sorts as *kinds* [21, 22, 23, 24, 25].

The systems all have the same axiom, namely

$$\mathsf{Prop} : \mathsf{Type}.$$

They all have the following *general rules* in common:

(start) *If* $x \notin FV(\Gamma)$

$$\frac{\Gamma \ \vdash \ A : s}{\Gamma, x : A \ \vdash \ x : A}$$

(weakening) *If* $x \notin FV(\Gamma)$

$$\frac{\Gamma \ \vdash \ A : B \quad \Gamma \ \vdash \ C : s}{\Gamma, x : C \ \vdash \ A : B}$$

(application)

$$\frac{\Gamma \ \vdash \ M : (\forall x : A)B \quad \Gamma \ \vdash \ N : A}{\Gamma \ \vdash \ MN : [N/x]B}$$

(abstraction) *If* $x \notin FV(\Gamma)$

$$\frac{\Gamma, x : A \ \vdash \ M : B \quad \Gamma \ \vdash \ (\forall x : A)B : s}{\Gamma \ \vdash \ \lambda x : A \,.\, M : (\forall x : A)B}$$

(conversion)

$$\frac{\Gamma \ \vdash \ A : B \quad \Gamma \ \vdash \ B' : s \quad B =_\beta B'}{\Gamma \ \vdash \ A : B'}$$

The systems are differentiated by the following *specific rules*:

(ss' rule) *If* $x \notin FV(\Gamma)$

$$\frac{\Gamma \ \vdash \ A : s \quad \Gamma, x : A \ \vdash \ B : s'}{\Gamma \ \vdash \ (\forall x : A)B : s'} \quad ,$$

where the different systems depend on the possible values of s and s'. Here are some of the specific systems:

- $\lambda \rightarrow$, related to simple type assignment: Both s and s' must be Prop.
- $\lambda 2$, related to Second order typed λ-calculus: s' must be Prop.
- λP, related to AUT-QE and LF: s must be Prop.

- $\lambda\omega$, related to Girard's $F\omega$: If s is Prop, so is s'.
- λC, Calculus of constructions: s and s' can both be either kind.

HOL, which is Church's simple type theory [9], which is not in the λ-cube, is a subsystem of λC. λC is the strongest system in the λ-cube; all the other systems are subsystems of it. Similarly, $\lambda \rightarrow$ is a subsystem of all other systems in the cube.

These systems all have some advantages for the purposes we are considering in this paper:

- They can all be proved consistent by proving a strong normalization theorem. The proof of this theorem varies from one system to another; the proof for stronger systems is a stronger proof than that for weaker systems. As Gödel's Second Theorem tells us, each of these proofs uses means of proof which cannot be formalized in the system involved.
- They all have a small number of primitive postulates, which means that when they are implemented there are few places for programming errors to occur.[2]

They do have one feature which might be considered a disadvantage: they are impredicative. But all the known predicative systems suffer from the disadvantage that they have a large number of primitive postulates. It thus appears that we can have the advantages of a small number of postulates (with the consequence of a relatively small chance of an error in programming) or else predicativity. My own personal view is that a small number of primitive postulates is more important than predicativity, especially since in practice, most clients for such systems will probably never have heard of predicativity.

The formulation given here is the one given by Barendregt as a Pure Type System (PTS) [3], and, in the case of the calculus of constructions, is much closer to the original formulation of Coquand and Huet [11] than the formulation I used in my previous papers on this subject [21, 22, 23, 24, 25]. The main difference is that before an environment Γ can appear in a

[2]Compare this to the system Nuprl [10], which has over one hundred primitive postulates, each of which must be programmed separately in implementation.

deduction in the formulation given here, it must be proved well-formed by proving that

$$\Gamma \vdash \mathsf{Prop} : \mathsf{Type},$$

whereas in the formulation I formerly used, being well-formed was shown to be necessary for the discharge of assumptions by conditions on the rules corresponding to (abstraction) and the (ss' rule). For

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$$

to be well-formed means that

- The variable x_i does not occur free in $A_1, A_2, \ldots, A_i$ (but it may occur free in $A_{i+1}, \ldots, A_n$), and
- $x_1 : A_1, x_2 : A_2, \ldots x_{i-1} : A_{i-1} \vdash A_i : s$ for some sort s.

From now on, all environments will be assumed to be well-formed.

2 Representing Logic With Equality

Logic is represented in these typed systems by the *propositions-as-types* interpretation, which is also known as the *Curry-Howard isomorphism*. [14]. The idea is that the types are interpretated as formulas or propositions, and the terms are interpreted as proofs or deductions. It is not hard to see that deductions in any of these systems using the rules for (application) and (abstraction) follow the constructions of the terms involved, and the other rules play the auxiliary role of determining when these two rules, and especially the rule for (abstraction), can be legally applied.

The notation $A \to B$ is used for the type $(\forall x : A)B$ when x does not occur free in B. As a type, $A \to B$ is the type of functions whose arguments are in type A and whose values are in type B. When $A : \mathsf{Prop}$ and $B : \mathsf{Prop}$, the type $A \to B$ will be interpreted as the implication from A to B, and the notation $A \supset B$ will often be used for this. The rule (application) gives us modus ponens, and the rule (abstraction) gives us implication introduction in the usual sense of natural deduction.

The other connectives and quantifiers are defined to the extent that they can be defined in the various systems of the λ-cube:[3]

[3] See [22, §6].

Definition 2 The connectives and quantifiers are defined as follows:

- If $A : \mathsf{Prop}$ and $B : \mathsf{Prop}$, use $A \wedge B$ for $(\forall w : \mathsf{Prop})((A \supset B \supset w) \supset w)$. The terms of type $A \wedge B$ are pairs with projections, and their types give us the usual properties of conjunction in natural deduction.

- If $A : \mathsf{Prop}$ and $B : \mathsf{Prop}$ use $A \vee B$ for

  $$(\forall w : \mathsf{Prop})((A \supset w) \supset ((B \supset w) \supset w))$$

  Terms of type $A \vee B$ are disjoint unions with injections, and their types give us the usual properties of disjunction in natural deduction.

- Define $\mathsf{void} \equiv \bot \equiv (\forall x : \mathsf{Prop})x$. If $A : \mathsf{Prop}$, use $\neg A$ for $A \supset \bot$. This gives us the usual properties of intuitionistic negation. Furthermore, it follows from the strong normalization theorem that there is no closed term M such that $\vdash M : \mathsf{void}$.

- If $A : \mathsf{Prop}$ and $x : A \vdash B : \mathsf{Prop}$, then use $(\exists x : A)B$ for

  $$(\forall w : \mathsf{Prop})((\forall x : A)(B \to w) \to w).$$

  Terms of type $(\exists x : A)B$ are pairs (differently typed from those of type $A \wedge B$) with a left projection but (for technical reasons) no right projection. The types involved give us the usual natural deduction rules for the existential quantifier.

- If $A : s$, $M : A$ and $N : A$, use $M =_A N$ for

  $$(\forall z : A \to \mathsf{Prop})(zM \supset zN).$$

  This is called *Leibniz equality*. It is easy to prove that it satisfies the reflexive, symmetric, and transitive laws and that equals under this equality can always be replaced by equals.

With these definitions, we have a higher-order intuitionistic logic.
We can also define a *Boolean* type:

$$\mathsf{Bool} \equiv (\forall u : \mathsf{Prop})(u \to u \to u),$$

$$\mathsf{T} \equiv \lambda u : \mathsf{Prop} \,.\, \lambda x : u \,.\, \lambda y : u \,.\, x,$$

$$\mathsf{F} \equiv \lambda u : \mathsf{Prop} \,.\, \lambda x : u \,.\, \lambda y : u \,.\, y.$$

Here T and F have distinct normal forms. The usual truth functions are easy to define, but none of them are identified with the connectives and quantifiers defined above.

3 Adding Unproved Assumptions

The higher-order intuitionistic logic with equality that we have seen above is not enough for practical theorem provers and proof assistants. We will also need to allow for new postulates in the form of unproved assumptions. Adding such assumptions is easy: to make A an assumption, just add $c : A$ as a new assumption, where c is a new atomic constant. New atomic constants can be obtained from variables by simply deciding not to make substitutions for them. In an implementation, variables and constants are just identifiers anyway, and any computer implementation allows for any number of those.

However, adding unproved assumptions in this way can cause problems with consistency. The basic system is consistent, as noted above. However, it is easy to come up with a set of new assumptions which leads to a contradiction:

$$c_1 : \mathsf{Prop}, c_2 : c_1, c_3 : \neg c_1.$$

The contradiction that follows from these assumptions does not in any way negate the consistency that follows from the strong normalization theorem. Nevertheless, it is important to avoid them in any practical application.

Definition 3 A set Γ of assumptions is *consistent* if there is no term M such that $\Gamma \vdash M : \bot$. This is equivalent to: there is no term N such that $\Gamma, x : \mathsf{Prop} \vdash N : x$, where $x \notin \mathrm{FV}(\Gamma)$.

We now want to prove consistent sets of assumptions that are useful in practical applications of theorem-proving. The best way to do this is to prove these consistency results for the strongest system we are considering, namely the calculus of constructions. The results so obtained will then follow for all those weaker systems in which the assumptions can be typed. In some weaker systems, the results can be used to justify assuming new assumptions. For example, in $\lambda \rightarrow$, our definition of conjunction cannot be typed. However, we can add a new constant Λ, take $A \wedge B$ as an abbreviation for ΛAB, and then add the following assumptions:

- $\Lambda : \mathsf{Prop} \rightarrow \mathsf{Prop} \rightarrow \mathsf{Prop}$.
- $\mathsf{and.in} : \lambda x : \mathsf{Prop} \,.\, \lambda y : \mathsf{Prop} \,.\, x \rightarrow y \rightarrow \Lambda xy$.
- $\mathsf{and.left} : \lambda x : \mathsf{Prop} \,.\, \lambda y : \mathsf{Prop} \,.\, \Lambda xy \rightarrow x$.

- and.right : $\lambda x : \mathsf{Prop} \,.\, \lambda y : \mathsf{Prop} \,.\, \Lambda xy \to y$.

The fact that the resulting system can be interpreted in a stronger system (the calculus of constructions) in which these assumptions are interpreted as provable results shows that adding them to $\lambda \to$ will not lead to contradiction.

4 The Consistency of Unproved Assumptions

In order to obtain the consistency results we desire, we need to consider normalizing deductions as well as terms.

Deduction Normalization A deduction of the form

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash (\forall x : A)B : s}{\Gamma \vdash \lambda x : A \,.\, M : (\forall x : A)B}\,(\text{abstraction})}{\Gamma \vdash \lambda x : A \,.\, M : (\forall x : C)D}\,(\text{conversion}) \qquad \Gamma \vdash N : C}{\Gamma \vdash (\lambda x : A \,.\, M)N : [N/x]D}\,(\text{application})$$

where $x \notin \mathrm{FV}(\Gamma, A)$, $A =_\beta C$, and $B =_\beta D$, reduces to

$$\dfrac{\dfrac{\Gamma, x : A \vdash M : B \quad \dfrac{\Gamma \vdash N : C}{\Gamma \vdash N : A}\,(\text{conversion})}{\Gamma \vdash [N/x]M : [N/x]B}\,(\text{substitution lemma})}{\Gamma \vdash [N/x]M : [N/x]D.}\,(\text{conversion})$$

The substitution lemma is proved for all systems of the λ-cube in [3, Lemma 5.1.11].

Strong normalization holds for deductions with this reduction relation.

Note that this kind of reduction step is related to the kind of implication reduction step considered by Prawitz in [20]:

$$\dfrac{\dfrac{\begin{array}{c}1\\[A]\\\mathcal{D}_1\\B\end{array}}{A \supset B}\supset \mathrm{I}-1 \quad \begin{array}{c}\mathcal{D}_2\\A\end{array}}{\begin{array}{c}B\\\mathcal{D}_3\\C\end{array}}\supset \mathrm{E}$$

reduces to

$$\begin{array}{c}\mathcal{D}_2\\A\\\mathcal{D}_1\\B\\\mathcal{D}_3\\C.\end{array}$$

Here, as part of the reduction step, the deduction

$$\begin{array}{c}\mathcal{D}_2\\A\end{array}$$

is placed above all occurrences of A as an undischarged assumption in

$$\begin{array}{c}A\\\mathcal{D}_1\\B.\end{array}$$

This part of the reduction step corresponds to the use of the Substitution Lemma above. But the use of the substitution lemma is more complicated. This is because in order to make use of an assumption in a deduction, it must be made the conclusion of the (start) rule, and so, unlike ordinary natural deduction, the assumption does not occur at a leaf of the tree. In fact, the only way to make as assumption occur at the top of a branch is to *define* a branch to start at the conclusion of an inference by rule (start).

One of the main features of normalized deductions used in [20] is that if the deduction ends in an elimination rule, then the only inferences which can occur in the main branch are other inferences by elimination rules, and so the assumption at the top of the main branch is not discharged. This form of reasoning applies here as well, provided that we understand that the assumption at the top of the main branch is on the right side of the conclusion of an inference by (start). The rules are written here so that the main branch is always the leftmost branch, so the main branch will sometimes be referred to as the "left branch."

There are certain kinds of inferences that we want to avoid in the main branch of a deduction if we are to have consistency. Let Γ be

$$A : \mathsf{Prop}, w : (\forall z : A \to \mathsf{Prop})(zN), x : \mathsf{Prop}.$$

What we want to avoid is the following:

$$\dfrac{\dfrac{\Gamma \vdash w : (\forall z : A \to \mathsf{Prop})(zN) \quad \Gamma \vdash \lambda y : A \,.\, \vdots\; x : A \to \mathsf{Prop}}{\Gamma \vdash w(\lambda y : A \,.\, x) : (\lambda y : A \,.\, x)N}\ (\text{application})}{\Gamma \vdash w(\lambda y : A \,.\, x) : x.}\ (\text{conversion})$$

In order to avoid this, we define a class of assumptions that cannot occur undischarged at the top of the main branch in which such an inference occurs:

Definition 4 Let Γ be an environment of the form

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n.$$

For each i, let A_i convert to

$$(\forall y_{i1} : B_{i1})(\forall y_{i2} : B_{i2}) \ldots (\forall y_{im_i} : B_{im_i})S_i,$$

where S_i, the *tail* of A_i, does not convert to a term of the form $(\forall z : C)D$.[4] Then Γ is *strongly consistent* if for each A_i for which $m_i > 0$ and S_i converts to $z_i M_{i1} M_{i2} \ldots M_{il}$, if z_i is a variable, then the tail of its type does not convert to Prop.

A strongly consistent environment is consistent.

This definition is very weak. A strongly consistent environment cannot contain any types of the form $A \wedge B$, $A \vee B$, $\bot$, $\neg A$, $(\exists x : A)B$, or $M =_A N$.

Although a strongly consistent environment cannot contain negations of types, there are consistent environments that can. See [22, Definition 27].

Theorem 1 *Let Γ_1 be a well-formed environment in which each type is the negation of an equation between terms with distinct normal forms, and let Γ_2 be strongly consistent. Then if, for $B : s$ and a closed term R,*

$$\Gamma_1, \Gamma_2 \vdash R : M =_B N,$$

then $M =_\beta N$.

[4] It can be shown that every type in a well-formed environment converts a term of this form.

This is [22, Theorem 20]. The idea behind the proof is to show that the deduction ending in $R : M =_B N$ must have the form

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \Gamma_2, z : B \to \mathsf{Prop}, u : zM \ \vdash \ R_1 : zM \quad \vdots \quad \vdots}{\Gamma_1, \Gamma_2, z : B \to \mathsf{Prop}, u : zM \ \vdash \ R_1 : zN}\ (\text{conversion}) \quad \vdots}{\Gamma_1, \Gamma_2, z : B \to \mathsf{Prop} \ \vdash \ \lambda u : zM \,.\, R_1 : zM \supset zN}\ (\text{abstraction}) \quad \vdots}{\Gamma_1, \Gamma_2 \ \vdash \ \lambda z : B \to \mathsf{Prop} \,.\, \lambda u : zM \,.\, R_1 : (\forall z : B \to \mathsf{Prop})(zM \supset zN).}\ (\text{abstraction})$$

In this deduction, the inference by (conversion) is only valid if $M =_* N$ and R_1 is u. In showing that the deduction is of this form, it is necessary to show in each case that the formula in question cannot be the conclusion of an inference by (application), and this is shown by reasoning about the assumption at the top of the main branch if it is: the assumptions $z : B \to \mathsf{Prop}$ and $u : zM$ can both be in a strongly consistent environment, and it cannot be any of those, and if it is in Γ_1, then the minor premise would have to have a conclusion of the form $M' =_{B'} N'$, contrary to the hypothesis that we are dealing with the shortest deduction of that kind.

Note that a consequence of this theorem is the following corollary:

Corollary 1.1 *If Γ is a strongly consistent environment, then*

$$\Gamma, c : \neg(\mathsf{T} =_{\mathsf{Bool}} \mathsf{F})$$

is consistent.

Note also that the theorem identifies Leibniz equality with conversion.

S. Berardi [5] assumes that Leibniz equality is extensional in the sense that, for terms M and N of type $(\forall x_1 : A_1)(\forall x_2 : A_2) \ldots (\forall x_m : A_m)\mathsf{Prop}$,

$$(\forall x_1 : A_1)(\forall x_2 : A_2) \ldots (\forall x_m : A_m)(Mx_1x_2 \ldots x_m \leftrightarrow Nx_1x_2 \ldots x_m) \supset M =_B N,$$

where B is $(\forall x_1 : A_1)(\forall x_2 : A_2) \ldots (\forall x_m : A_m)\mathsf{Prop}$ and $A \leftrightarrow B$ is $(A \supset B) \wedge (B \supset A)$. This assumption has unusual consequences when $m = 0$. For let A be any inhabited type. Then A is provable. But so is $A \to A$. It follows from this assumption that $A =_{\mathsf{Prop}} (A \to A)$ is provable. It follows from this that A is a model of the untyped λ-calculus, and so any function of type $A \to A$ has a fixed point. Since the type N defined in the next section is inhabited and since the successor function $\boldsymbol{\sigma}$ has type $\mathsf{N} \to \mathsf{N}$, $\boldsymbol{\sigma}$ has a fixed point! This is a highly unusual and counterintuitive result, and it is

undesirable in small trusted theorem provers and proof assistants. For this reason, this assumption of extensionality will not be made here.

Classical logic can be obtained by adding the assumption

$$\mathsf{cl} : (\forall u : \mathsf{Prop})(\neg\ \neg\ u \supset u),$$

where cl is a new constant. This assumption can be proved consistent with strongly consistent assumptions plus negations of equations between distinct normal forms by means of a variation of the double-negation translation. See [22, Corollary 22.1].

5 Representing Arithmetic

It is standard to represent arithmetic in this system with the following definitions:

- $\mathsf{N} \equiv (\forall A : \mathsf{Prop})((A \to A) \to (A \to A))$
- $\mathbf{0} \equiv \lambda A : \mathsf{Prop}\ .\ \lambda x : A \to A\ .\ \lambda y : A\ .\ y$
- $\boldsymbol{\sigma} \equiv \lambda u : \mathsf{N}\ .\ \lambda A : \mathsf{Prop}\ .\ \lambda x : A \to A\ .\ \lambda y : A\ .\ x(uAxy)$

Here, N is the type of the natural numbers, $\mathbf{0}$ is the number zero, and $\boldsymbol{\sigma}$ is the successor function. Then a natural number n is represented by

$$\mathsf{n} =_\beta \lambda A : \mathsf{Prop}\ .\ \lambda x : A \to A\ .\ \lambda y : A\ .\ \underbrace{x(x(\ldots(x}_{n}\, y)\ldots))$$

It is possible to define $\boldsymbol{\pi}$ so that

$$\begin{aligned} \boldsymbol{\pi}\mathbf{0} &=_\beta \mathbf{0} \\ \boldsymbol{\pi}(\boldsymbol{\sigma}\mathsf{n}) &=_\beta \mathsf{n} \end{aligned}$$

Using this $\boldsymbol{\pi}$, it is possible to define R so that if $A : \mathsf{Prop}$, $M : A$, and $N : \mathsf{N} \to A \to A$,

$$\begin{aligned} \mathsf{R}MN\mathbf{0} &=_\beta M \\ \mathsf{R}MN(\boldsymbol{\sigma}\mathsf{n}) &=_\beta N\mathsf{n}(\mathsf{R}MN\mathsf{n}) \end{aligned}$$

We can prove

$$\begin{aligned} \vdash \quad & \mathsf{N} : \mathsf{Prop} \\ \vdash \quad & \mathbf{0} : \mathsf{N} \\ \vdash \quad & \boldsymbol{\sigma} : \mathsf{N} \to \mathsf{N} \end{aligned}$$

This is an example of an inductively defined datatype. This particular representation is an example of the kind of datatype treated in [8, 6].

We would naturally like to prove mathematical induction for N, which is

$$(\forall A : \mathsf{N} \to \mathsf{Prop})((\forall u : \mathsf{N})(Au \supset A(\boldsymbol{\sigma} u)) \supset A\mathbf{0} \supset (\forall x : \mathsf{N})(Ax)),$$

but there is a major problem about this. It says that every term of type N is Leibniz equal to a term representing a natural number. However, the theorem we proved above identifies Leibniz equality with conversion, which in these systems is β-conversion. But there is a term in type N that does not β-convert to a term representing a natural number: $\lambda A : \mathsf{Prop} \,.\, \lambda x : A \,.\, x$. The term does η-convert to a term representing a natural number, but we are not using η-conversion here.[5] Pfenning and Paulin-Mohring [19] give an example of a recursive datatype represented this way in which there is a term in the type which does not β- or η-convert to anything constructed from the constructors of the datatype. For this reason, it appears that this form of mathematical induction is false in these systems and that it would be a mistake to assume it.

Instead, we can get the principle of mathematical induction by using Dedekind's definition of the natural numbers [13]:

$$\mathcal{N} \equiv \lambda n : \mathsf{N} \,.\, (\forall A : \mathsf{N} \to \mathsf{Prop})((\forall m : \mathsf{N})(Am \supset A(\boldsymbol{\sigma} m)) \supset A\mathbf{0} \supset An)$$

We can prove

$$\begin{aligned} \vdash \quad & \mathcal{N} : \mathsf{N} \to \mathsf{Prop}, \\ \vdash \quad & \mathcal{N}\mathbf{0}, \\ \vdash \quad & (\forall n : \mathsf{N})(\mathcal{N}n \supset \mathcal{N}(\boldsymbol{\sigma} n)), \\ \vdash \quad & (\forall A : \mathsf{N} \to \mathsf{Prop})((\forall m : \mathsf{N})(Am \supset A(\boldsymbol{\sigma} m)) \supset A\mathbf{0} \supset, \\ & \qquad (\forall n : \mathsf{N})(\mathcal{N}n \supset An)). \end{aligned}$$

[5] The rule for η-reduction is that $\lambda x : A \,.\, Ux \rhd U$ if x is not free in U.

Thus, by relativizing the quantifiers to $\mathcal{N}$, we can obtain the use of mathematical induction. Furthermore, we have used only definitions; there are no new unproved assumptions. Thus, if we are working within an environment known to be consistent, we retain consistency.

We can extend these results to the other Peano axioms. For example, using $\boldsymbol{\pi}$ we can prove

$$\vdash\ (\forall n : \mathsf{N})(\forall m : \mathsf{N})(\mathcal{N}n \supset \mathcal{N}m \supset \boldsymbol{\sigma}n =_{\mathsf{N}} \boldsymbol{\sigma}m \supset n =_{\mathsf{N}} m).$$

Also, using $\mathsf{Bool} : \mathsf{Prop}$, $\mathsf{T} : \mathsf{Bool}$, and $\lambda n : \mathsf{N}\ .\ \lambda x : \mathsf{Bool}\ .\ \mathsf{F} : \mathsf{N} \to \mathsf{Bool} \to \mathsf{Bool}$, all of which are provable, we can define

$$\mathsf{Iszero} \equiv \mathsf{RT}(\lambda n : \mathsf{N}\ .\ \lambda x : \mathsf{Bool}\ .\ \mathsf{F})$$

Then for $\mathsf{n} : \mathsf{N}$, since $\vdash\ \mathcal{N}\mathsf{n}$,

$$\begin{aligned} \mathsf{Iszero}\ \mathbf{0} &=_{\beta} \mathsf{T}, \\ \mathsf{Iszero}\ (\boldsymbol{\sigma}\mathsf{n}) &=_{\beta} \mathsf{F}. \end{aligned}$$

Hence, we can prove

$$\mathsf{bool} : \neg\ \mathsf{T} =_{\mathsf{Bool}} \mathsf{F} \vdash\ (\forall n : N)(\mathcal{N}n \supset \neg\ \boldsymbol{\sigma}n =_{\mathsf{N}} \mathbf{0}).$$

This means that *arithmetic in a typed system is consistent.*

In a recent paper [24], this is extended to a large class of abstract recursively defined data types. But not all: only those for which the type of each constructor has the form

$$A_1 \to (A_2 \to \ldots (A_n \to D) \ldots)$$

where D is the type of the database and each A_i is either D or is the type of another such database or is a variable of type Prop. All of these datatypes are covariant in the sense of [1]. A method of representing non-covariant datatypes is given by Appel and McAllester in [2]. I believe that a more natural method is to extend Dedekind's definition to arbitrary datatypes by defining a predicate for any set of constructors which says that an object (of a suitable type) is an object of the datatype if it satisfies every predicate which is closed under the constructors.[6] To deal with recursive definitions

[6]As I write this, I have not yet found a general way of writing this predicate, but I have succeeded for an example which is not covariant.

(i.e., definitions of recursive functions), I propose to use the logic to prove that valid definitions do, indeed, define functions. For the case of definitions corresponding to primitive recursive functions of natural numbers, I propose to adapt a proof that primitive recursive definitions define total functions of natural numbers that was given by Lorenzen [18] and Kalmár [17].[7]

This approach to inductive definitions, as well as the approach of [1, 2], has the advantage that it is carried out by making definitions, and not by adding new rules to the type theory as is done in [12, 26, 7]. When extra rules are added, there are existential committments that follow from them, and this means that restrictions must be placed on the rules in advance in order to avoid inconsistency. When new definitions are used without new unproved assumptions, there is more flexibility. And with the approach I am advocating, merely extending Dedekind's approach to defining the natural numbers does not, in itself involve any existential committments. The worst that can happen if a definition is given that does not correspond to a real inductive structure is that it will be impossible to prove that anything satisfies the definition, and this causes no problems with consistency. Furthermore, when the proof of Lorenzen and Kalmár that primitive recursive definitions define total functions is adapted to this kind of theory, the theorem that is proved has an existential hypothesis, so this theorem only proves that something exists if it can be proved that there is an object of the right type satisfying the inductive definition. With this approach, the logic itself will take care of these problems, and this, in my opinion, makes the approach more general than the others.

6 Representing Sets as Predicates

In [15, Chapter 5] and [16], Huet proposed to represent a significant part of elementary set theory by means of predicates. The idea is to take a type $U : \mathsf{Prop}$[8] as a universe and to define Set_U to be $U \to \mathsf{Prop}$. Then if $A : \mathsf{Set}_U$, $x \in A$ is just Ax. Furthermore, for $P : \mathsf{Prop}$, the set $\{x : U | P\}$ is $\lambda x : U \,.\, P$. In addition, we have the following definitions:

$$A \subseteq B \;\equiv\; (\forall x : U)(x \in A \supset x \in B),$$

[7]The adaptation of this proof seems to work in the case of the example mentioned in the preceding footnote.

[8]The definition will also work if $U : \mathsf{Type}$, but this is not needed here.

$$
\begin{array}{rcl}
A =_{\mathrm{ex}} B & \equiv & (A \subseteq B) \wedge (B \subseteq A), \\
\emptyset & \equiv & \{x : U \mid \bot\}, \\
\{x\} & \equiv & \{y : U | y =_U x\}, \\
A \cap B & \equiv & \{x : U \mid x \in A \wedge x \in B\}, \\
A \cup B & \equiv & \{x : U \mid x \in A \vee x \in B\}, \\
\sim A & \equiv & \{x : U \mid \neg\, x \in A\}, \\
\mathcal{P}A & \equiv & \lambda B : \mathsf{Set}_U \,.\, B \subseteq A, \\
\mathsf{Class}_U & \equiv & \mathsf{Set}_U \to \mathsf{Prop}.
\end{array}
$$

Note that if $A : \mathsf{Set}_U$, then $\mathcal{P}A : \mathsf{Class}_U$. Other definitions involving classes are

$$
\begin{array}{rcl}
\bigcap C & \equiv & \{x : U | (\forall A : \mathsf{Set}_U)(CA \to x \in A)\}, \\
\bigcup C & \equiv & \{x : U | (\exists A : \mathsf{Set}_U)(CA \wedge x \in A)\}.
\end{array}
$$

If $A, B : \mathsf{Set}_U$, we can represent the collection of functions from A to B by

$$\lambda f : U \to U \,.\, (\forall x : U)(x \in A \supset fx \in B).$$

This is a lot of set theory. In [22, Remark 17], it is shown that all the axioms of the constructive set theory IZF [4, p. 164] except for $\in$-induction and power set are provable in this representation. The axiom of $\in$-induction is the constructive replacement for the axiom of foundation, whose role is to prevent infinite descending $\in$-chains, but here such chains are prevented by the type structure. So the important axiom that cannot be proved here is the axiom of power set. But power sets may be taken any finite number of times, and this is enough for many practical purposes.

7 Conclusion

We have seen that with only two unproved assumptions, namely

(1) $$c_1 \;:\; \neg(\mathsf{T} =_{\mathsf{Bool}} \mathsf{F}),$$

(2) $$\mathsf{cl} \;:\; (\forall u : \mathsf{Prop})(\neg\neg u \supset u),$$

we can obtain classical arithmetic and a good deal of set theory. Furthermore, the combination of these assumptions has been proved consistent. This shows that we have systems which

- are small,
- have few primitive postulates,
- are provably consistent,
- are sufficient for a lot of mathematical reasoning.

These systems should be useful for a number of applications of theorem proving.

References

[1] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM Symposium on the Principles of Programming Languages*, pages 243–253, 2000.

[2] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. Technical Report TR-629-00, Princeton University, 2000.

[3] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[4] M. Beeson. *Foundations of Constructive Mathematics*. Springer, Berlin, 1985.

[5] Stefano Berardi. *Type Dependence and Constructive Mathematics*. PhD thesis, Universita di Torino, 1989.

[6] Stefano Berardi. Encoding of data types in pur construction calculus: a semantic justification. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 30–60. Cambridge University Press, Cambridge, 1993.

[7] F. Blanqui. The calculus of algebraic and inductive constructions. Technical report, DEA Sémantique, preuve et Programmation, 1998.

[8] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39(2–3):135–154, 1985.

[9] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[10] R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[11] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[12] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88: Proceedings of the International Conference on Computer Logic held in Tallinn, December 12–16, 1988*, volume 417 of *Springer Lecture Notes in Computer Science*, pages 50–66, 1990.

[13] Richard Dedekind. *Was sind und was sollen die Zahen?* Friedr. Vieweg & Sohn, Braunschweig, 10th edition, 1965. 1st edition, 1887.

[14] W. A. Howard. The formulae-as-types notion of construction. In J. Roger Hindley and Jonathan P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, New York, 1980. A version of this paper was privately circulated in 1969.

[15] Gérard Huet. Formal structures for computation and deduction. Course Notes, Carnegie-Mellon University, First Edition, May 1986.

[16] Gérard Huet. Induction principles formalized in the calculus of constructions. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT '87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, March 23–27, 1987. Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP '87)*, volume 249 of *Lecture Notes in Computer Science*, pages 276–286, Berlin, 1987. Springer-Verlag.

[17] László Kalmár. On the possibility of definition by recursion. *Acta Szeged*, 9(4):227–232, 1940.

[18] Paul Lorenzen. Die Definition durch vollständige Induktion. *Monatsch. Math. u. Phys.*, 47:356–358, 1939.

[19] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics: 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29–April 1, 1989, Proceedings*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, 1989.

[20] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Stockholm, Göteborg, and Uppsala, 1965.

[21] Jonathan P. Seldin. Coquand's calculus of constructions: a mathematical foundation for a proof development system. *Formal Aspects of Computing*, 4:425–441, 1992.

[22] Jonathan P. Seldin. On the proof theory of Coquand's calculus of constructions. *Annals of Pure and Applied Logic*, 83:23–101, 1997.

[23] Jonathan P. Seldin. A Gentzen-style sequent calculus of constructions with expansion rules. *Theoretical Computer Science*, 243:199–215, 2000.

[24] Jonathan P. Seldin. On lists and other abstract data types in the calculus of constructions. *Mathematical Structures in Computer Science*, 10:261–276, 2000. Special issue in honor of J. Lambek.

[25] Jonathan P. Seldin. Extensional set equality in the calculus of constructions. *Journal of Logic and Computation*, 11(3):483–493, 2001. Presented at Festival Workshop in Foundations and Computations held at Heriot-Watt University, Edinburgh, 16-18 July, 2000.

[26] Benjamin Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris 7, 1994.