Finite axiomatizations
of inductive and inductive-recursive definitions

Peter Dybjer and Anton Setzer
May 14, 1998

Abstract
We first present a finite axiomatization of strictly positive inductive
types in the simply typed lambda calculus. Then we show how this axiom-
atization can be modified to encompass simultaneous inductive-recursive
definitions in intuitionistic type theory. A version of this has been imple-
mented in the Half system which is based on Martin-Löf’s logical frame-
work.

1 Introduction
The present note summarizes a presentation to be given at the Workshop on
We use Martin-Löf’s logical framework as a metalanguage for axiomatizing
inductive definitions in the simply typed lambda calculus. We also show how to
genralize this axiomatization to the case of inductive-recursive definitions in the
lambda calculus with dependent types. The reader is referred to the full paper
[7] for a more complete account focussing on induction-recursion. Related papers
discussing inductive definitions in intuitionistic type theory include Backhouse
[1, 2], Coquand and Paulin [3], Dybjer [5, 4], and Paulin [?] For an introduction
to inductive-recursive definitions see Dybjer [6].

2 Inductive definitions in the simply typed lambda
calculus

2.1 Inductive types as initial algebras
Let us first consider the question of how to formalize inductive types in the
setting of the simply typed $\lambda$-calculus. We shall consider generalized inductive
definitions of types given by a finite number of constructors

\[ \text{intro} : \Phi U \to U, \]

where $\Phi$ is strictly positive in the following restricted sense

- The constant functor $\Phi D = 1$ is strictly positive. This is the base case
  corresponding to an introduction rule with no premises.
• If $\Psi$ is strictly positive and $A$ is a (small) type, then $\Phi D = A \times \Psi D$ is strictly positive. This corresponds to the addition of a non-recursive premise.

• If $\Psi$ is strictly positive and $B$ is a (small) type, then $\Phi D = (B \to D) \times \Psi D$ is strictly positive. This corresponds to the addition of a recursive premise, where $B$ corresponds to the hypotheses of this premise in a generalized inductive definition.

Note that all occurrences of $U$ in $\Phi U$ are strictly positive in the standard sense that $U$ does not occur to the left of an arrow in $\Phi U$.

Recall that such inductive types can be captured categorically as initial $\Phi$-algebras for an endofunctor $\Phi$ on a category of types. An initial $\Phi$-algebra is an arrow

$$\Phi U \xrightarrow{\text{intro}} U$$

such that for any other $\Phi$-algebra

$$\Phi D \xrightarrow{d} D$$

there is a unique arrow $T : U \to C$, such that the following diagram commutes

$$\begin{array}{ccc}
\Phi U & \xrightarrow{\text{intro}} & U \\
\Phi D & \xrightarrow{d} & D
\end{array}$$

$T$ is a function defined by iteration. In order to obtain definition by primitive or structural recursion we consider the following diagram:

$$\begin{array}{ccc}
\Phi U & \xrightarrow{\text{intro}} & U \\
\Phi U \times \Phi D & \xrightarrow{y \mapsto (y, R y)} & U \times D
\end{array}$$

and the equation

$$R (\text{intro } a) = d a (\Phi R a)$$

2.2 Type-theoretic formalization

Call inductive types "sets" and call a type generated from sets by $\to$, $\times$, and 1, a "small type".

To obtain a formal system for inductive types, we first inductively generate the type $\text{SP}$ of codes for strictly positive functors. To each code $\phi$ we associate an object part $\text{Arg}_\phi$ and the arrow part $\text{map}_\phi$ of the strictly positive functor
in question. Moreover, a function \( d : \text{Arg}_\phi D \rightarrow D \) for some \( \phi : \text{SP} \) is called a (possibly infinitary) \( D \)-operator.

With this new notation the initial algebra diagram becomes:

\[
\begin{array}{c}
\text{Arg}_\phi U \xrightarrow{\text{intro}} U \\
\downarrow \map_{\phi, U, D} T \quad \downarrow T \quad \downarrow \map_{\phi, D} d \\
\text{Arg}_\phi D \rightarrow D
\end{array}
\]

We thus have the following three formation rules:

- **SP type**
  
  \[
  \phi : \text{SP} \quad \text{D type} \\
  \text{Arg}_\phi D \quad \text{U, D type} \\
  \text{T : U} \rightarrow \text{D}
  \]

  \[
  \map_{\phi, U, D} T : \text{Arg}_\phi U \rightarrow \text{Arg}_\phi D
  \]

(We also have rules which state that \( \text{Arg}, \map, \) etc preserve equality, but we will not spell them out here.)

We have the following introduction rules for \( \text{SP} \):

- nil : \( \text{SP} \)
  
  \[
  \begin{array}{c}
  A : \text{stype} \quad \phi : \text{SP} \\
  \text{nonrec A} \phi : \text{SP} \\
  B : \text{stype} \quad \phi : \text{SP} \\
  \text{rec B} \phi : \text{SP}
  \end{array}
  \]

and the corresponding equality rules for \( \text{Arg} \):

- \( \text{Arg}_{\text{nil}} D = 1 \)
  
- \( \text{Arg}_{\text{nonrec A}} \phi D = A \times (\text{Arg}_\phi D) \)
  
- \( \text{Arg}_{\text{rec B}} \phi D = (B \rightarrow D) \times (\text{Arg}_\phi D) \)

and map

- \( \map_{\text{nil}, C, D} T * = * \)
  
- \( \map_{\text{nonrec A}, C, D} T (\gamma, a) = (\map_{\phi, C, D} T \gamma, a) \)
  
- \( \map_{\text{rec B}, C, D} T (\gamma, f) = (\map_{\phi, C, D} T \gamma, T \circ f) \)

An inductive type (a “set”) is now given by a finite list of codes for strictly positive operators, that is, by an object of \( \text{SP}^n \) for some number \( n \). We can now give the formation, introduction, elimination, and equality rules for the set \( U_\rho \), given by \( \rho : \text{SP}^n \).

We thus have the following formation rule for inductive types

- \( \rho : \text{SP}^n \)
  
- \( U_\rho : \text{set} \)
Moreover, we have the introduction rule

$$\rho : \text{SP}^n \quad i : N_n$$
$$\text{intro}_{\rho,i} : \text{Arg}_{\rho,i} U_\rho \rightarrow U_\rho$$

Let $\rho : \text{SP}^n$ and $D$ type be global premises in the elimination and equality rules.

The elimination rule for iteration is

$$d : (i : N_n) \rightarrow (\text{Arg}_{\rho,i} D) \rightarrow D$$
$$T_\rho d : U_\rho \rightarrow D$$

The equality rule for iteration is

$$d : (i : N_n) \rightarrow (\text{Arg}_{\rho,i} D) \rightarrow D \quad i : N_n \quad a : \text{Arg}_{\rho,i} U_\rho$$
$$T_\rho d \text{intro}_{\rho,i} a = d i (\text{map}_{(\rho,i)} U D) (T_\rho d) a : D$$

The elimination rule for primitive recursion is

$$d : (i : N_n) \rightarrow (\text{Arg}_{\rho,i} U) \rightarrow (\text{Arg}_{\rho,i} D) \rightarrow D$$
$$R_\rho d : U_\rho \rightarrow D$$

The equality rule for primitive recursion is

$$d : (i : N_n) \rightarrow (\text{Arg}_{\rho,i} U) \rightarrow (\text{Arg}_{\rho,i} D) \rightarrow D \quad i : N_n \quad a : \text{Arg}_{\rho,i} U_\rho$$
$$R_\rho d \text{intro}_{\rho,i} a = d i a (\text{map}_{(\rho,i)} U) (R_\rho d) a : D$$

3 Inductive-recursive definitions

3.1 Informal discussion

In the case of a simultaneous inductive-recursive definition the domain of the constructor intro may depend on $T : U \rightarrow D$ as well as on $U$. Therefore, it also (indirectly) depends on $D$. If we write out these new dependencies in the diagram for initial algebras we get the following tentative picture

$$\Phi D I$$
$$\Phi D \downarrow \text{intro} \downarrow$$
$$d$$
$$\Phi D \downarrow \text{R} \downarrow$$
$$D$$

Think of $D$ as a type of “semantic” objects and of $d : \Phi D \rightarrow D$ as a (possibly infinitary) “semantic” operation with $\Phi D$ as the generalized “arity” of $d$. $U$ is a universe of codes for objects in $D$ and $T : T \rightarrow D$ is the decoding function. The (possibly infinitary) constructor intro is the syntactic reflection of $d : \Phi D \rightarrow D$.

Note that $\Phi$ is no longer an endofunctor on the category of types, but rather consists of a triple, since what corresponds to the object part now is split up into

- a “semantic” part $D \mapsto \Phi D$ which returns the source (generalized arity) of $d$;
• and a “syntactic” part \( D, I, R \mapsto \Phi DIR \) which returns the source of intro.

Moreover, the modified class (of codes for such triples) \( SP = SP D \) depends on the semantic type \( D \) and needs to be defined simultaneously with the decoding function – which returns the codomain of a semantic operation. \( SP \) and – together specify what it means to be a (possibly infinitary) operation on \( D \) in the framework of dependent types.

### 3.2 Type-theoretic formalization

We shall now give the formal rules for inductive-recursive definitions. Such a definition is always parameterized with respect to a type \( D \): a particular inductive-recursive definition defines a universe for a finite number of \( D \)-operators.

So first we need to define what a \( D \)-operator is. To this end we simultaneously introduce the type \( SP D \) of codes for arities (codomains) of \( D \)-operators and the associated decoding function

\[
\begin{array}{c|c}
\text{D type} & \phi : SP D \\
\hline
\text{Arg} \phi & \text{type}
\end{array}
\]

which returns the codomain (coded by some \( \phi : SP D \)) of a \( D \)-operator. Note that, since the type of \( \phi \) depends on \( D \), we can no longer have \( D \) as an argument to \( \text{Arg} \) which comes after \( \phi \), as was the case for the simply typed case where \( \phi \) indeed coded a functor.

\[
\begin{array}{c}
\text{arg}_{D, \delta} U T \xrightarrow{\text{intro}} U \\
\text{map}_{D, \delta} U T \xrightarrow{d} D
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{D type} & \delta : SP_D \\
\hline
\text{Arg}_{D, \delta} & \text{type}
\end{array}
\]

\[
\begin{array}{ccc}
\text{D type} & \delta : SP D & U \text{ set} \\
\hline
\text{arg}_{D, \delta} U T & T : U \to D
\end{array}
\]

\[
\begin{array}{ccc}
\text{D type} & \delta : SP_D & U \text{ set} \\
\hline
\text{map}_{D, \delta} U T : (\text{arg}_{D, \delta} U T) \to \text{Arg}_{D, \delta}
\end{array}
\]

From now on \( D \text{ type} \) is a global premise.

\[
\begin{array}{c}
nil : SP_D \\
A \text{ stype} \xrightarrow{\delta : A \to SP_D} \nonrec A \delta : SP_D \\
B \text{ stype} \xrightarrow{\delta : (B \to D) \to SP_D} \rec B \delta : SP_D
\end{array}
\]

5
\[
\begin{align*}
\text{Arg}_{D, \text{nil}} &= 1 \\
\text{Arg}_{D, \text{nonrec } A} &= (x : A) \times \text{Arg}_{D, \delta x} \\
\text{Arg}_{D, \text{rec } B} &= (f : B \to D) \times \text{Arg}_{D, \delta f}
\end{align*}
\]

\[
\begin{align*}
\text{arg}_{D, \text{nil}} UT &= 1 \\
\text{arg}_{D, \text{nonrec } A} UT &= (x : A) \times (\text{arg}_{D, \delta x} UT) \\
\text{arg}_{D, \text{rec } B} UT &= (f : B \to U) \times (\text{arg}_{D, \delta (f \circ f)} UT)
\end{align*}
\]

\[
\begin{align*}
\text{map}_{D, \text{nil}} UT * &= * \\
\text{map}_{D, \text{nonrec } A} UT (a, \gamma) &= (a, \text{map}_{D, \delta a} UT \gamma) \\
\text{map}_{D, \text{rec } B} UT (f, \gamma) &= (f \circ f, \text{map}_{D, \delta (f \circ f)} UT \gamma)
\end{align*}
\]

References


