Recursive Families of Inductive Types

Venanzio Capretta

Computing Science Institute, University of Nijmegen
Postbus 9010, 6500 GL Nijmegen, The Netherlands
e-mail: venanzio@cs.kun.nl
telephone: +31+24+3652647, fax: +31+24+3553450

Abstract. Families of inductive types defined by recursion arise in the formalization of mathematical theories. An example is the family of term algebras on the type of signatures. Type theory does not allow the direct definition of such families. We state the problem abstractly by defining a notion, *strong positivity*, that characterizes these families. Then we investigate its solutions. First, we construct a model using wellorderings. Second, we use an extension of type theory, implemented in the proof tool Coq, to construct another model that does not have extensionality problems. Finally, we apply the two level approach: We internalize inductive definitions, so that we can manipulate them and reason about them inside type theory.

1 Introduction

In type theory we can define a new inductive type by giving its constructors (or introduction rules). For example, we define the types of natural numbers, binary trees, and lists over a type A as

$$\mathbb{N}: \quad \frac{}{0 \colon \mathbb{N}} \quad \frac{n \colon \mathbb{N}}{(S\ n) \colon \mathbb{N}}, \qquad \mathbb{T}: \quad \frac{}{\mathsf{leaf} \colon \mathbb{T}} \quad \frac{x_1 \colon \mathbb{T} \quad x_2 \colon \mathbb{T}}{\mathsf{node}(x_1, x_2) \colon \mathbb{T}}, \quad \text{and}$$

$$\mathsf{List}(A): \quad \frac{}{\mathsf{nil}_A \colon \mathsf{List}(A)} \quad \frac{a \colon A \quad l \colon \mathsf{List}(A)}{\mathsf{cons}_A(a, l) \colon \mathsf{List}(A)},$$

respectively.

Consider the family $T \colon \mathbb{N} \to *$ ($*$ indicates the type of all small types, or sets) of inductive types indexed on the natural numbers:

$$\begin{array}{llll}
T_0: & & & \dfrac{}{c_0 \colon T_0} \\[2ex]
T_1: & & \dfrac{}{c_1 \colon T_1} & \dfrac{x \colon T_1}{c_0(x) \colon T_1} \\[2ex]
T_2: & \dfrac{}{c_2 \colon T_2} & \dfrac{x \colon T_2}{c_1(x) \colon T_2} & \dfrac{x_1 \colon T_2 \quad x_2 \colon T_2}{c_0(x_1, x_2) \colon T_2} \\[2ex]
\vdots & & &
\end{array} \tag{1}$$

Every new type in the family is defined by a new constant and by the constructors of the previous type in the hierarchy with an extra recursive argument. Intuitively T_n is the type of trees with branching degree at most n. In the standard formulation of inductive types this definition is not allowed: The constructors and their types must be given directly at the moment of definition of the inductive type, whereas the number of constructors of T_n and their types are defined by recursion on n.

Families of this kind have not only theoretical interest. They arise in the course of formalization of mathematics in a proof tool. I first encountered them when I was working on the formalization of Universal Algebra in Coq (see [6] and [7]). The family of term algebras on the type of signatures is one of them. The type of single-sorted signatures is $\mathsf{Sig} := \mathsf{List}(\mathbb{N})$. Given a signature $\sigma := [a_1, \ldots, a_n]$, the type of terms over σ is defined by

$$\mathsf{Term}_\sigma : \quad \frac{t_{11} : \mathsf{Term}_\sigma \ \cdots \ t_{1a_1} : \mathsf{Term}_\sigma}{(\mathsf{f}_1 \ t_{11} \ \cdots t_{1a_1}) : \mathsf{Term}_\sigma} \quad \cdots \quad \frac{t_{n1} : \mathsf{Term}_\sigma \ \cdots \ t_{na_n} : \mathsf{Term}_\sigma}{(\mathsf{f}_n \ t_{n1} \ \cdots t_{na_n}) : \mathsf{Term}_\sigma}.$$

(One of the a_i's must be 0, so that Term_σ is nonempty.) We cannot obtain the family $\mathsf{Term} : \mathsf{Sig} \to *$ directly with an inductive definition, because the number and arity of the constructors depend on the signature σ: They are not fixed for the whole family. The situation is even more complicated when we consider many-sorted signatures, which require families of mutual inductive types. In [6] we used Martin-Löf's W types to solve this instance of the problem. Here we formulate the general problem, we show that W types still provide a good model, but also propose a better solution (which, however, requires an extension of type theory). You can see the details of its application to many-sorted algebras in [7].

In Section 3 we formulate the general problem: We propose an extension of the notion of strictly positive operator, which is used to determine the admissibility of inductive definitions, using *positive type pointers*—that is, terms that specify the positive occurrence of parameters in recursive definitions.

In Section 4 we represent inductive types using Martin-Löf's type constructor for wellorderings (W types) (see [14, 15] and chapter 15 of [18]), extending the work by Dybjer [10]. This solution has the disadvantage that structurally equal elements of a W-type are not always convertible, thus making the W-type representation only *extensionally* isomorphic to the desired inductive type.

Alternatively, we can exploit the extension of the positivity condition implemented in the system Coq and described by Gimenez in [12]. It allows an inductive definition to inherit a positive occurrence of a type variable from another inductive definition. To use this construction in our case, we need to give a translation of our recursive family of operators into an inductive family. In Section 5 we give such translation and we use it to solve our problem.

Finally, in Section 6 we use the two level approach (see [4], [5], [13] and [3]): Positivity is a metapredicate; that is, it is not expressed inside type theory but is an external syntactic property of type operators. This means that we cannot reason about positive operators and inductive definitions inside type theory. We internalize it by defining a type-theoretic predicate $\mathsf{Positive}$ expressing the

metaproperty. We define also a type of codes for inductive types and associate a code to every proof of an instance of the predicate Positive. We define a function that associates a type to every code. Now we solve our problem by first, constructing a family of positive operators by recursion; second, proving their positivity inside type theory; third, obtaining the corresponding family of codes; finally, instantiating the codes to types. This last method has been completely formalized in the proof assistant Coq [2].

2 Inductive types

We work in a type theory that is at least as expressive as the Pure Type System $\lambda P\bar{\omega}$ (see [1]): There are two sorts of types, $*$ for small types and $\Box$ for large types. Sort $*$ is an element of $\Box$. Moreover, we have sum and Σ types, which can be considered as special cases of inductive types, which we define later in this section. In $\lambda P\bar{\omega}$ every small type $T\colon *$ has an isomorphic version in $\Box$. For simplicity we identify the two; in other words, we consider $*$ and $\Box$ as the first two steps in a cumulative hierarchy of type universes. When we write type expressions that mix the two sorts, as $T \times *$ or $T + *$, the version of T in $\Box$ is used. Note, however, that if $*$ is impredicative (for example, if we work in the Calculus of Constructions) not all elements of $*$ can have a representation in $\Box$, because this would lead to Girard's paradox (see [8]). Only if impredicativity was not used in the definition of the type, we can consider it as an element of $\Box$. When we use small types in $\Box$ constructions we assume that this condition is satisfied without saying it (as supported by the Coq implementation).

We use the notation $t[x]$ to denote a term t in which a variable x may occur. Thus t and $t[x]$ denote the same term, but in the second expression we stress the dependence on x. Do not confuse this notation with $(f\ x)$, which denotes the application of a function f to x. If s is a term of the same type as x, $t[s]$ denotes the result of the substitution $t[x := s]$.

In *extensional* type theory inductive types can be implemented as fixed points of type operators (see [17]). We are working in *intensional* type theory, in which inductive types are recursively defined by constructors. Following [9], [19], [21] and [23] an inductive type I is defined by a list of constructors:

$$\begin{array}{l}\mathsf{inductive}\ I\ [\overrightarrow{X : S}]\colon (z_1 : Q_1)\cdots(z_n : Q_n)* := \\ \quad c_1\ :\ (x_1 : P_{11})\cdots(x_{k_1} : P_{1k_1})(I\ M_{11}\cdots M_{1m}) \\ \quad \vdots \\ \quad c_n\ :\ (x_1 : P_{n1})\cdots(x_{k_1} : P_{nk_1})(I\ M_{n1}\cdots M_{nm}) \\ \mathsf{end},\end{array} \tag{2}$$

where I does not occur in the Ss, Qs and Ms and occurs only strictly positively in the Ps. $\overrightarrow{X}$ is a list of general parameters of I (such as the parameter X in $\mathsf{List}(X)$). See one of the cited references or chapter 4 of the Coq manual [2] for the definition of strict positivity and for the other rules.

If the types of the constructors do not use dependent product—that is, they are in the form $P_{i1} \to \cdots \to P_{ik_i} \to I$—we can use the alternative formulation of inductive types as fixed points of strictly positive type operators (see, for example, [10]). It is less intuitive but simpler for theoretical purposes, so we adopt it. Every strictly positive operator $X : * \vdash \Phi[X] : *$ has a functorial extension, which, for $X, Y : *$, maps every $f : X \to Y$ to a function $\Phi[f] : \Phi[X] \to \Phi[Y]$; preserving identities and composition (see [9] and [20]). This condition is sufficient to formulate the rules for inductive types (Matthes [16] gives an extension of *system F* in which this is the only condition required for inductive types). In the next sections we consider extensions of the positivity condition that still have the functorial property. The rules for inductive types are then the same as in the following definition, with the corresponding property replacing *strictly positive*.

Definition 1. *Let $X : * \vdash \Phi[X] : *$ be a strictly positive operator. The* inductive type $\mu_X(\Phi)$ *is defined by the following rules (where we write I for $\mu_X(\Phi)$):*

formation $I : *$

introduction $$\frac{y : \Phi[I]}{(\mu\text{-intro } y) : I}$$

elimination $$\frac{x : I \vdash (P\ x) : * \quad z : \Phi[(\Sigma\ I\ P)] \vdash u : (P\ (\mu\text{-intro } (\Phi[\pi_1]\ z)))}{(\mu\text{-ind } [z]u) : (x : I)(P\ x)}$$

conversion $(\mu\text{-ind } [z]u\ (\mu\text{-intro } y)) \rightsquigarrow u[(\Phi[[x]\langle x, (\mu\text{-ind } [z]u\ x)\rangle]\ y)]$

We use this formulation to define our inductive types, since they are all non-dependent, but we use the notation of Formula 2 when it is intuitively clearer and when we need to define types whose constructors belong to dependent product types.

If the elimination predicate P is a constant type T, we obtain the recursion principle; if, furthermore, the recursion term u does not depend on the induction arguments, we obtain the iteration principle:

$$\frac{T : * \quad z : \Phi[I \times T] \vdash u : T}{(\mu\text{-rec } [z]u) : I \to T} \quad \text{and} \quad \frac{T : * \quad z : \Phi[T] \vdash u : T}{(\mu\text{-it } [z]u) : I \to T}.$$

It is well known that the recursion and iteration principles are equivalent, whereas the full induction principle is a proper extension of them (see, for example, [11] or [20]).

The types of natural numbers, binary trees, and lists over a type A can be defined as $\mathbb{N} := \mu_X(\mathbb{N}_1 + X)$, where $\mathbb{N}_1$ is the type with only one element 0_1; $\mathbb{T} := \mu_X(\mathbb{N}_1 + X \times X)$; and $\mathsf{List}(A) := \mathbb{N}_1 + A \times X$, respectively. Their constructors can be defined in terms of the single constructor μ-intro:

$$\begin{aligned} 0 &:= (\mu\text{-intro } (\mathsf{inl}\ 0_1)), & S &:= [n](\mu\text{-intro } (\mathsf{inr}\ n)); \\ \mathsf{leaf} &:= (\mu\text{-intro } (\mathsf{inl}\ 0_1)), & \mathsf{node} &:= [x_1, x_2](\mu\text{-intro } (\mathsf{inr}\ \langle x_1, x_2\rangle)); \\ \mathsf{nil} &:= (\mu\text{-intro } (\mathsf{inl}\ 0_1)), & \mathsf{cons} &:= [a, l](\mu\text{-intro } (\mathsf{inr}\ \langle a, l\rangle)). \end{aligned}$$

The problem that we consider here is: Given a family of type operators $\Phi\colon A \to (* \to *)$ such that every element of it is strictly positive, can we construct the corresponding family of inductive types? Observe that it is not possible to characterize such families in a decidable way. In fact, for every function $f : \mathbb{N} \to \mathbb{N}$ we can associate such a family:

$$\begin{array}{l}\Phi\colon \mathbb{N} \to (* \to *) \\ (\Phi\ n\ X) = \begin{cases} X & \text{if } (f\ n) = 0 \\ X \to X & \text{otherwise.}\end{cases}\end{array}$$

Deciding whether every element of this family is strictly positive is equivalent to deciding whether f is constantly 0. Since, in type theory, every primitive recursive function on the natural numbers is definable, we would be able to decide whether any such function is constantly 0, which is notoriously impossible.

The following section gives a decidable characterization of some of these families, which is wide enough for the examples that we are considering.

3 Families of inductive types defined by strong elimination

Strong elimination is the elimination rule for inductive types in which the elimination predicate is allowed to be *big*; that is, we can have an elimination predicate $x\colon I \vdash (P\ x)\colon \Box$. If $*$ is impredicative, strong elimination results in inconsistency (see [8]). Nevertheless, it can still be admitted if the inductive type I is defined without the use of impredicativity—that is, as already mentioned, if there is a type in $\Box$ isomorphic to it. In such a case we allow strong elimination. This form of strong elimination is supported in Coq. We use strong elimination only in the form of iteration over the type $*$:

$$\frac{Z\colon \Phi[*] \vdash W[Z]\colon *}{(\mu\text{-it}\ [Z]W)\colon I \to *}.$$

We recast Example 1 as

$$\begin{array}{l}X\colon * \vdash \Psi[X]\colon \mathbb{N} \to * \\ \qquad\quad \Psi_0 \quad\ := \mathbb{N}_1 \\ \qquad\quad \Psi_{(S\ n)} := \mathbb{N}_1 + X \times \Psi_n.\end{array}$$

To be completely formal, we must write

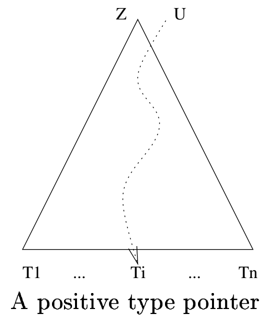
$$\frac{\begin{array}{l}X\colon *, Z\colon \mathbb{N}_1 + * \vdash W[X,Z]\colon * \\ \qquad := \mathsf{Case}\ Z\ \mathsf{of} \\ \qquad\qquad (\mathsf{inl}\ \ _) \Rightarrow \mathbb{N}_1 \\ \qquad\qquad (\mathsf{inr}\ \ R) \Rightarrow \mathbb{N}_1 + X \times R \\ \qquad \mathsf{end}\end{array}}{X\colon * \vdash \Psi[X] := (\mu\text{-it}\ [Z]W)\colon \mathbb{N} \to *}. \qquad (3)$$

The desired family of inductive types is now specified by $T_n := \mu_X(\Psi_n)$. Unfortunately, this is not an allowed definition, since Ψ does not satisfy the strict-positivity condition: Although Ψ_n reduces to a strictly positive operator for each numeral n, Ψ_x does not if $x\colon \mathbb{N}$ is a variable. Therefore, we cannot define the family $T\colon \mathbb{N} \to *$, even if every member of it is individually definable.

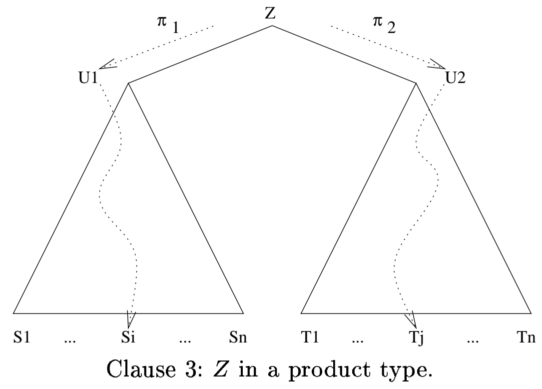
The case of term algebras over single-sorted signatures is similar. The operator associated to a signature $\sigma := [a_1, \ldots, a_n]$ is $\Psi_\sigma[X] := X^{a_1} + \cdots + X^{a_n}$. We can define first a single component $(n\colon \mathbb{N}, X : * \vdash X^n\colon *)$ by strong elimination on the natural numbers and then Ψ_σ by strong elimination on $\mathsf{List}(\mathbb{N})$. The type of terms associated with the signature σ is then $\mathsf{Term}_\sigma := \mu_X(\Psi_\sigma)$. As in the previous example this definition is not allowed in the standard implementation of inductive types, because Ψ_σ does not satisfy the strict-positivity condition.

Our purpose is to find ways to define families of inductive types in type theory. The first step is a formal description of the problem—that is, an abstract characterization of the definitions we are looking for. If we consider the first of the preceding examples, we see that the reason why every single element of the family is strictly positive is that, in the recursive step of the definition, not only the type parameter X but the recursive call Ψ_n (or R in the formalized version) also occurs only strictly positively. It is enough to require that all such recursive calls occur strictly positively. Note, however, that, in the formal version of the definition, the recursive call is Z, not R, which is just a bound variable in the Case construction. It doesn't mean anything to say that Z occurs positively and the variable R does not actually occur in W (being bound). So we need a finer notion than strict positivity. The following concept of *positive type pointer* solves the problem. To understand it intuitively, consider a generic premise for a definition by strong elimination $(X\colon *, Z\colon \Theta[*] \vdash U[X,Z]\colon *)$ where Θ is a strictly positive operator. We want to define first the family of type operators $X\colon * \vdash \Psi[X] := (\mu\text{-}\mathsf{it}\ [Z]U)\colon \mu_Y(\Theta[Y]) \to *$ and then the family of inductive types $x\colon \mu_Y(\Theta[Y]) \vdash \mu_X((\Psi\ x))\colon *$. Imagine Z represented as a tree structure whose leaves are types representing recursive calls. We must require that each occurrence of a term pointing to such a leaf occurs only strictly positively in U.

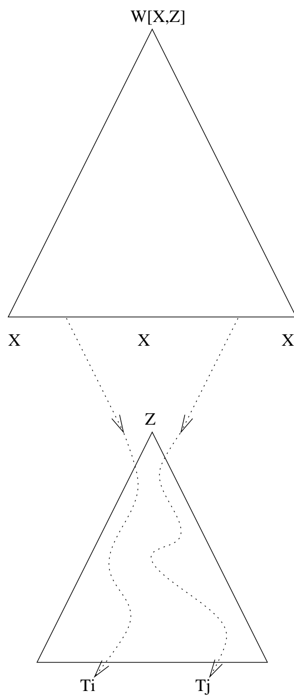
Figure 1 is a depiction of type pointers. It shows how a type pointer can be constructed for each of the type constructors that build new strongly positive operators. If Z is in a product type (clause 3), it is a pair, represented by a binary node with a subtree for each branch. A positive type pointer first chooses one of the components and then uses a positive type pointer for that component. If Z is in a function type (clause 5), the situation is similar, but the number of components is equal to the cardinality of the domain type (possibly infinite). If Z is in a sum type (clause 4), then it is in one of the two forms specified by the component types. A positive type pointer must take into account both possibilities, so it prescribes a type pointer for each of the two components. Therefore, the picture for clause 3 shows two positive type pointers corresponding to the two components, whereas the picture for clause 4 shows only one positive type pointer that consists of two components. The picture for clause 8 shows how a positive type pointer is used in a recursive definition of a family of strongly



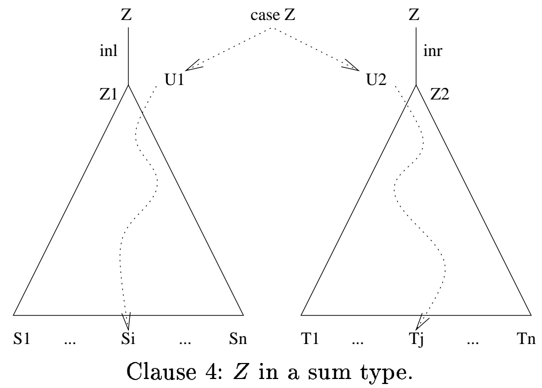
A positive type pointer



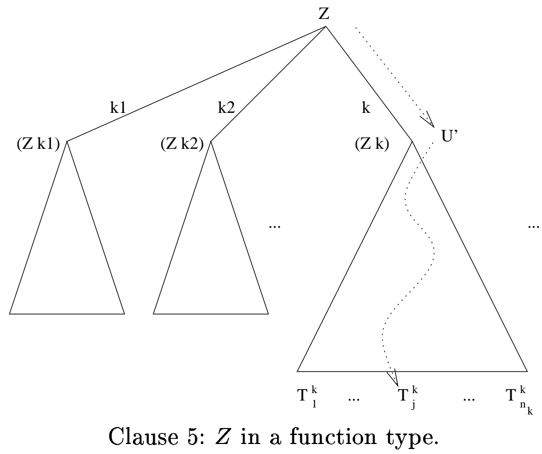
Clause 3: Z in a product type.



Clause 8: a strongly positive family.



Clause 4: Z in a sum type.



Clause 5: Z in a function type.

Fig. 1. Illustration of Definition 2: the use of positive type pointers in the definition of families of strongly positive operators.

positive operators. The term $W[X, Z]$ is the iterator of the recursive definition. It contains some direct occurrences of the variable X and some recursive calls, here indicated by the leaves T_i and T_j of the iteration variable Z. When T_i and T_j are replaced with the values of the recursive call, new occurrences of X appear. The requirement that $W[X, Z]$, besides being strongly positive in X, is also a positive type pointer in Z causes all the new occurrences of X to be strictly positive.

Definition 2. *A type operator* $(X : * \vdash \Phi[X] : *)$ *that can be lifted to kinds* $(X : \Box \vdash \Phi[X] : \Box)$ *is* strongly positive *and a term* $(Z : \Phi[*] \vdash U[Z] : *)$ *is a* positive type pointer *for* Φ *if they satisfy the following recursive clauses.*

1. *If* K *is a type that does not depend on* X *(that is,* X *does not occur free in* K*), then* $\Phi[X] = K$ *is strongly positive and* $Z : K \vdash K : *$ *is a positive type pointer for* Φ.
2. *If* $\Phi[X] = X$ *then* Φ *is strongly positive and* $Z : * \vdash Z : *$ *is a positive type pointer for* Φ.
3. *If* $\Phi[X] = \Phi_1[X] \times \Phi_2[X]$ *and* Φ_1 *and* Φ_2 *are strongly positive, then* Φ *is strongly positive and if* $Z_1 : \Phi_1[*] \vdash U_1[Z_1] : *$ *and* $Z_2 : \Phi_2[*] \vdash U_2[Z_2] : *$ *are positive type pointers for* Φ_1 *and* Φ_2*, respectively, then*
$$Z : \Phi_1[*] \times \Phi_2[*] \vdash (U_1\ (\pi_1\ Z)) : * \quad \textit{and}$$
$$Z : \Phi_1[*] \times \Phi_2[*] \vdash (U_2\ (\pi_2\ Z)) : *$$
are positive type pointers for Φ.
4. *If* $\Phi[X] = \Phi_1[X] + \Phi_2[X]$ *and* Φ_1 *and* Φ_2 *are strongly positive, then* Φ *is strongly positive and if* $Z_1 : \Phi_1[*] \vdash U_1[Z_1] : *$ *and* $Z_2 : \Phi_2[*] \vdash U_2[Z_2] : *$ *are positive type pointers for* Φ_1 *and* Φ_2*, respectively, then*
$$Z : \Phi_1[*] + \Phi_2[*] \vdash \mathsf{Case}\ Z\ \mathsf{of}\ (\mathsf{inl}\ Z_1) \Rightarrow U_1[Z_1] \mid (\mathsf{inr}\ Z_2) \Rightarrow U_2[Z_2]\ \mathsf{end} : *$$
is a positive type pointer for Φ.
5. *If* $\Phi[X] = K \to \Phi'[X]$*, where* K *is a type that does not depend on* X*, and* Φ' *is a strongly positive type operator, then* Φ *also is strongly positive and if* $Z' : \Phi'[*] \vdash U'[Z'] : *$ *is a positive type pointer for* Φ'*, then, for every* $k : K$*,*
$$Z : K \to \Phi'[*] \vdash U'[(Z\ k)] : *$$
is a positive type pointer for Φ.
6. *Suppose* $t : A_1 + A_2$ *for types* A_1 *and* $A_2 : *$*. If* $\Phi[X] = (\mathsf{Case}\ t\ \mathsf{of}\ (\mathsf{inl}\ x_1) \Rightarrow \Phi_1[x_1, X] \mid (\mathsf{inl}\ x_2) \Rightarrow \Phi_2[x_2, X]\ \mathsf{end})$ *and* Φ_1 *and* Φ_2 *are strongly positive, then* Φ *also is strongly positive.*
7. *If* Φ *and* Ψ *are strongly positive operators and* $Z : \Phi[*] \vdash U[Z] : *$ *is a positive type pointer for* Φ*, then* $Z : \Phi[*] \vdash \Psi[U[Z]] : *$ *also is a positive type pointer for* Φ.
8. *If* $Y : * \vdash \Theta[Y] : *$ *is a strongly positive type operator,* $I = \mu_Y(\Theta)$*, and* $X : *, Z : \Theta[*] \vdash W[X, Z] : *$ *is a positive type pointer for* Θ *with respect to* Z *and is strongly positive with respect to* X*, then every element of the family* $X : * \vdash \Phi[X] := (\mu\text{-}\mathsf{it}\ [Z]W[X, Z]) : I \to *$ *is strongly positive; that is, for every* $i : I$*,* $X : * \vdash (\Psi[X]\ i) : *$ *is a strongly positive type operator.*

Clause 7 may seem too restrictive because we do not consider the possibility that different positive type pointers for Φ may be used in Ψ. For example, if $\Psi[Y] = Y \times Y$, we may want to define the type pointer $X : * \vdash U_1[X] \times U_2[X]$ where U_1 and U_2 are different positive type pointers for Φ. In that case we should modify Φ such that it becomes an operator on two parameters, $Y_1, Y_2 : * \vdash Y_1 \times Y_2$, and then apply clause 7 twice, the first type substituting $U_1[X]$ for Y_1, the second time substituting $U_2[X]$ for Y_2. This can be done in all similar situations.

We do not include a definition of positive type pointer corresponding to the strongly positive operator obtained in clause 8. This further complication is not necessary to define the families of types in which we are interested.

The definition of strongly positive type operator coincides with the definition of strictly positive type operator but for the last clause, which allows the definition of families of strongly positive type operators by recursion, using a positive type pointer as the recursion term.

For example, consider the family of type operators defined in Formula (3). We want to prove that $X : * \vdash (\Psi[X]\ n)$ is strongly positive for every $n : \mathbb{N}$. Since $\mathbb{N} = \mu_Y(\Theta[Y])$ with $\Theta[Y] = \mathbb{N}_1 + Y$, we can use clause 8 of Definition 2. We have to prove that $X : *, Z : \mathbb{N}_1 + * \vdash W[X, Z] : *$ is a strongly positive type operator with respect to X and a positive type pointer for Θ with respect to Z. The first property follows from clause 6 and the easily verifiable fact that the two branches of the Case definition are strongly positive with respect to X (they are actually strictly positive). The second follows from clause 4 with $\Phi_1[Y] = \mathbb{N}_1$ and $U_1[Z_1] = \mathbb{N}_1$ (positive type pointer by clause 1), $\Phi_2[Y] = Y$ and $U_2[Z_2] = \mathbb{N}_1 + X \times Z_2$ (positive type pointer by clause 7, with $\Psi[V] = \mathbb{N}_1 + X \times V$, and clause 2).

It follows that, in the system extended by Definition 2, we can define the family of inductive types $T := [n : \mathbb{N}]\mu_X(\Psi[X]\ n) : \mathbb{N} \to *$.

Definition 2 does not add new inductive types to the system, but simply allows us to collect types in new families.

Theorem 1. *Every closed type $\mu_X(\Phi)$ definable by Definition 2 is definable by Definition 1 also.*

Proof We must prove that every strongly positive operator $X : * \vdash \Phi[X] : *$ in which no free variable except X occurs, is strictly positive (or, better, reduces to a strictly positive one). The proof is by induction on the number of times clause 8 of Definition 2 is used. We don't need to consider the other clauses, since they are the same in the definition of strict positivity. Suppose then that Φ has been obtained by clause 8—that is, $\Phi = (\Psi\ a)$, where Ψ is as in clause 8 and a is a closed term of type I. We assume that a is in normal form (otherwise we normalize it). We prove that $(\Psi\ a)$ is strictly positive by induction on the set of closed terms of I in normal form. (Note that this is structural induction external to type theory, and not an internal application of the elimination rule. This explains why a must be a *closed* term for it to work.) Suppose $a = (\mu\text{-}\mathsf{intro}\ b)$

with $b\colon \Theta[I]$ closed. Then

$$\begin{aligned}(\Psi\ a) &= (\Psi\ (\mu\text{-intro}\ b)) \\ &= (\mu\text{-it}\ [Z]W[X,Z]\ (\mu\text{-intro}\ b)) \\ &\leadsto W[X, (\Theta[(\mu\text{-it}\ [Z]W[X,Z])]\ b)] \\ &= W[X, (\Theta[\Phi[X]]\ b)]\end{aligned}$$

The term $(\Theta[\Phi[X]]\ b)$ can be represented as a tree isomorphic to the structure tree of a and whose leaves are in the form $(\Psi\ c)$, with c an element of I structurally simpler than a. By induction hypothesis, for all recursive occurrences of elements of I in b (that is, the elements of I that are structurally simpler that a), the corresponding elements of the family Ψ are strictly positive. Since $(\Psi\ a)$ is strictly-positively constructed from such occurrences by the type pointer W (this is the main property of the notion of positive type pointer and can be proved straightforwardly for every clause of Definition 2), it is also strictly positive. □

4 Wellorderings

In the previous section we proposed an extension of the notion of inductive type. We see now that, without extending type theory, we can encode the desired types and families as wellorderings. Wellorderings (also called W types) are types of trees specified by a type of nodes A and, for every element a of A, a type of branches $(B\ a)$. This means that every node labelled with the element a has as many branches as the elements of $(B\ a)$.

Wellorderings were introduced by Martin-Löf [14, 15] and used by Dybjer [10] to encode all inductive types obtained from strictly positive operators. Here we extend Dybjer's construction to strongly positive operators.

Definition 3. *Let $A\colon *$ and $B\colon A \to *$. The type $\mathsf{W}(A,B)$ is defined by the rules*

formation $\mathsf{W}(A,B)\colon *$

introduction

$$\frac{a\colon A \quad f\colon (B\ a) \to \mathsf{W}(A,B)}{(\mathsf{sup}\ a\ f)\colon \mathsf{W}(A,B)}$$

elimination *Let $P\colon \mathsf{W}(A,B) \to *$, then*

$$\frac{\begin{array}{l}x\colon A, y\colon (B\ x) \to \mathsf{W}(A,B), z\colon (u\colon (B\ x))(P\ (y\ u)) \\ \vdash e[x,y,z]\colon (P\ (\mathsf{sup}\ x\ y))\end{array}}{(\mathsf{W}\text{-}\mathsf{ind}\ [x,y,z]e)\colon (w\colon \mathsf{W}(A,B))(P\ w)}$$

conversion

$$\begin{aligned}&(\mathsf{W}\text{-}\mathsf{ind}\ [x,y,z]e\ (\mathsf{sup}\ a\ f)) \\ &\quad\leadsto e[a,f,[u\colon (B\ a)](\mathsf{W}\text{-}\mathsf{ind}\ [x,y,z]e\ (f\ u))]\end{aligned}$$

Wellorderings can be realized in type theory with the standard implementation of inductive types. Using Formula 2 we can define the W constructor as

$$\begin{array}{l}\mathsf{inductive}\ \mathsf{W}\ [A:*, B: A \to *]: * := \\ \quad \mathsf{sup}:\ (x:A)((B\ x) \to \mathsf{W}(A,B)) \to \mathsf{W}(A,B) \\ \mathsf{end}.\end{array}$$

Dybjer showed in [10] that every strictly positive operator has an initial algebra constructed by a W type. This result holds if we take an extensional equality on the W type—that is, if we consider two elements $(\mathsf{sup}\ a_1\ f_1)$ and $(\mathsf{sup}\ a_2\ f_2)$ of $\mathsf{W}(A,B)$ equal if a_1 and a_2 are convertible and if $(f_1\ b) = (f_2\ b)$ for every $b\colon (B\ a_1)$. In intensional type theory, which is the one we use, the second condition is not equivalent to the convertibility of f_1 and f_2. For this reason, when we use W types, we have to deal explicitly with extensional equality. They are, therefore, more cumbersome than direct inductive definitions. Once we have stressed this drawback, we can extend Dybjer's result to strongly positive operators.

Theorem 2. *For every strongly positive operator $X\colon * \vdash \Phi[X]\colon *$ there exist $A\colon *$ and $B\colon A \to *$ such that $\mathsf{W}(A,B)$ is an initial algebra of Φ. (For a formal definition of initial algebras of type operators see, for example, [11] or [20].)*

Proof The proof is by induction on the structure of Φ as in Dybjer [10]. Our Definition 2 contains two extra clauses that are not present in Dybjer's definition: clauses 6 and 8. Let us see how Dybjer's proof can be extended to include them.

Clause 6 is easily treated by defining A and B by cases on the term t in the definition of Ψ and using the recursive results for the branches of the Case expression. (See the following example.)

If Ψ is obtained by clause 8 we define the families $A\colon I \to *$ and $B\colon (x\colon I)A_x \to *$ by recursion on I. Given $x = (\mu\text{-}\mathsf{intro}\ y)\colon I$, we assume by inductive hypothesis that A and B are defined for all the recursive occurrences of elements of I in y. We define the new A_x and B_x, by using Dybjer's construction for the occurrences of X and of the recursive calls Z on $W[X,Z]$. Formally, using Dybjer's method recursively on the clauses of Definition 2, we can construct from W two families of operators W_A and W_B and then apply the iteration principle to obtain A and the induction principle to obtain B:

$$\frac{Z_A\colon \Theta[*] \vdash W_A[Z_A]\colon *}{A := (\mu\text{-}\mathsf{it}\ [Z_A]W_A)\colon I \to *},$$

$$\frac{Z_B\colon \Theta[(\Sigma\ I\ [i:I]A_i \to *)] \vdash W[Z_B]\colon A_{(\mu\text{-}\mathsf{intro}\ (\Theta[\pi_1]\ z))} \to *}{B := (\mu\text{-}\mathsf{ind}\ [Z_B]W_B)\colon (i\colon I)A_i \to *}.$$

Note the difference with the proof of Theorem 1: The assumption that a closed term $a\colon I$ is used was essential to that proof. That was necessary because we were proving an external predicate. But here we are constructing families of types internal to type theory, therefore we can use the elimination rule of type I to construct A (with elimination predicate $P_A = [x\colon I]*$) and B (with

elimination predicate $P_B = [x\colon I]A_x \to *$. Therefore A_x and B_x are defined also for a free variable $x\colon I$. □

This construction gives, in the case of the family of operators of Formula 3, the following families of As and Bs:

$$
\begin{array}{lll}
A\colon \mathbb{N} \to * & & B\colon (n\colon \mathbb{N})A_n \to * \\
\quad A_0 & := \mathbb{N}_1 & \quad (B_0 \quad _\ ) := \emptyset \\
\quad A_{(S\ n)} & := \mathbb{N}_1 + A_n & \quad (B_{(S\ n)}\ (\mathsf{inl}\ _)) := \emptyset \\
 & \quad (\cong \mathbb{N}_1 + \mathbb{N}_1 \times A_n) & \quad (B_{(S\ n)}\ (\mathsf{inr}\ a)) := \mathbb{N}_1 + (B_n\ a)
\end{array}
$$

The W construction for terms over a signature in Sig is described in [6], where it is extended to many-sorted signatures.

5 Recursive vs. Inductive families

We remarked that the W construction has the disadvantage that extensionally equal terms are not always convertible. This is unavoidable when we use transfinite types, but it could and should be avoided with finitary types. The solution proposed in this section exploits an extension of inductive types implemented in the proof tool Coq (see [12]). This consists in extending the notion of strict positivity to that of *positivity* by a clause that allows operators to inherit positive occurrences of a parameter X from inductive definitions.

Definition 4. *A type operator $X\colon * \vdash \Psi[X]\colon *$ is* positive *if it satisfies the clauses of the definition of strict positivity where we substitute "positive" for "strictly positive" everywhere, and the new clause*

> *X is positive in $(J\ t_1 \cdots t_m)$ if J is an inductive type and, for every term t_i, either X does not occur in t_i or X is positive in t_i, t_i instantiates a general parameter of J and this parameter is positive in the arguments of the constructors of J.*

To apply this construction to our case we first need to replace the recursive definition of a family of type operators with an inductive one. We illustrate the method with the example of Formula 3. The family Ψ was defined by recursion on the natural numbers. Instead we use the following inductive definition

$$
\begin{array}{l}
\mathsf{inductive}\ \mathsf{ind}(\Psi)[X : *]\colon \mathbb{N} \to * := \\
\quad \psi_0\ :\ \mathsf{ind}(\Psi)_0 \\
\quad \psi_1\ :\ (n\colon \mathbb{N})\mathsf{ind}(\Psi)_{(S\ n)} \\
\quad \psi_2\ :\ (n\colon \mathbb{N})X \to \mathsf{ind}(\Psi)_n \to \mathsf{ind}(\Psi)_{(S\ n)} \\
\mathsf{end}.
\end{array}
$$

(The constructors ψ_0 and ψ_1 could be unified in a single constructor $\psi_{01}\colon (n\colon \mathbb{N})\mathsf{ind}(\Psi)_n$, but we keep them separate to keep the parallel with the definition of Ψ in Formula 3.) X is a general parameter of $\mathsf{ind}(\Psi)$ and it is positive in the arguments of the constructors: It appears only as the type of the first

argument of the constructor ψ_2. It follows from the clause in Definition 4 that $X : * \vdash (\mathsf{ind}(\Psi)\ X\ n)$ is a positive type operator for every $n : \mathbb{N}$. In the type system of Coq such positive operators can be used in the definition of inductive types, thus the family $T := [n]\mu_X((\mathsf{ind}(\Psi)\ X\ n)) : \mathbb{N} \to *$ is admissible. Note that the condition expressed in clause 8 of Definition 2 by requiring W to be a positive type pointer corresponds to the fact that the recursive calls must occur positively in the definition of $\mathsf{ind}(\Psi)$. This translation can be done in general for every strongly positive operator.

Theorem 3. *For every strongly positive type operator $X : * \vdash \Phi[X] : *$ there exists a positive type operator $X : * \vdash \mathsf{ind}(\Phi)[X] : *$ such that, for every type $X : *$, $\Phi[X] \cong \mathsf{ind}(\Phi)[X]$.*

Proof As usual the relevant case is clause 8 of Definition 2. If $X : * \vdash \Psi[X] : I \to *$ is defined as in that clause, then we replace it with the inductive family

$$\begin{array}{l} \mathsf{inductive}\ \mathsf{ind}(\Psi)\ [X : *] : I \to * \\ \quad \psi : (y : \Theta[I])W[X, (\Theta[\mathsf{ind}(\Psi)]\ y)] \to \mathsf{ind}(\Psi)_{(\mu\text{-}\mathsf{intro}\ y)}, \\ \mathsf{end} \end{array}$$

which can be proved to be positive according to Definition 4, by induction on the proof that W is a positive type pointer. The general parameter X occurs only positively in the arguments of the constructor ψ because it occurs only positively in W (by induction hypothesis).

With this translation we get always inductive families with only one constructor. In practice it is intuitively easier to break it down into several constructors, as we did in the preceding example. □

6 Applying the two level approach to inductive types

The *two level approach* is a technique used for proof construction in type theory. A goal G is lifted to a syntactic level; that is, a term g, of a type $\mathsf{Goal} : *$ representing goals, is associated to G. Logical rules are reflected by functions or relations on Goal. To prove G we apply the functions or work with the relations on g. Once g is proved at the syntactic level, we can extract a proof of G.

The technique is described in [4] and in Ruys' thesis [22]. It was used by Boutin, who calls it *reflection*, to implement the `Ring` tactic in Coq [5]. Its furthest application consists in formalizing type theory inside type theory itself and use it to do metareasoning. This was partially done by Howe in [13] for Nuprl and by Barras and Werner in [3] for Coq.

We apply it to inductive definitions. First of all we define a type of codes for positive type operators PosOp. To every element $\phi : \mathsf{PosOp}$ we associate a positive type operator $(\mathsf{TypeOp}\ \phi) : * \to *$ and an inductive type $(\mathsf{IndType}\ \phi) : *$, using the technique of Section 5. Then we define an inductive predicate $\mathsf{Positive}$ on type operators, which is an internalization of the notion of strict positivity (note that we do not need to internalize strong positivity or positivity). We define a function that associates an element of PosOp to every operator $\Phi : * \to *$

and proof p: (Positive Φ). So we can define an inductive type by proving that the corresponding type operator is strictly positive. This can be done for the families of operators defined by recursion, hence solving our initial problem.

Definition 5. *The type* PosOp: $\square$ *is defined by the following introduction rules:*

$$\frac{K : *}{(\mathsf{op\text{-}const}\ K) : \mathsf{PosOp}} \qquad \frac{op_1 : \mathsf{PosOp} \quad op_2 : \mathsf{PosOp}}{(\mathsf{op\text{-}prod}\ op_1\ op_2) : \mathsf{PosOp}} \qquad \frac{K : * \quad op : \mathsf{PosOp}}{(\mathsf{op\text{-}fun}\ K\ op)}$$

$$\frac{}{\mathsf{op\text{-}id} : \mathsf{PosOp}} \qquad \frac{op_1 : \mathsf{PosOp} \quad op_2 : \mathsf{PosOp}}{(\mathsf{op\text{-}sum}\ op_1\ op_2) : \mathsf{PosOp}}.$$

We can associate an actual type operator to every element of PosOp, by recursion on it:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{TypeOp} : \mathsf{PosOp} \to * \to *} \\ \quad (\mathsf{op\text{-}const}\ K) & \Rightarrow [X : *]K \\ \quad \mathsf{op\text{-}id} & \Rightarrow [X : *]X \\ \quad (\mathsf{op\text{-}prod}\ op_1\ op_2) & \Rightarrow [X : *](\mathsf{TypeOp}\ op_1\ X) \times (\mathsf{TypeOp}\ op_2\ X) \\ \quad (\mathsf{op\text{-}sum}\ op_1\ op_2) & \Rightarrow [X : *](\mathsf{TypeOp}\ op_1\ X) + (\mathsf{TypeOp}\ op_2\ X) \\ \quad (\mathsf{op\text{-}fun}\ K\ op) & \Rightarrow [X : *]K \to (\mathsf{TypeOp}\ op\ X). \end{array}$$

Unfortunately this approach leads us to a dead end, since the family of operators TypeOp is strongly positive but not positive, being obtained by recursion.

We apply instead the technique of Section 5 to transform TypeOp from a recursive family to an inductive one satisfying the positivity condition:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathsf{inductive\ IndOp}\ [X : *] : \mathsf{PosOp} \to *} \\ \quad \mathsf{c}_{\mathsf{const}} & : & (K : *)K \to (\mathsf{IndOp}\ (\mathsf{op\text{-}const}\ K)) \\ \quad \mathsf{c}_{\mathsf{id}} & : & X \to (\mathsf{IndOp}\ \mathsf{op\text{-}id}) \\ \quad \mathsf{c}_{\mathsf{prod}} & : & (op_1, op_2 : \mathsf{PosOp})\,(\mathsf{IndOp}\ op_1) \to (\mathsf{IndOp}\ op_2) \\ & & \qquad \to (\mathsf{IndOp}\ (\mathsf{op\text{-}prod}\ op_1\ op_2)) \\ \quad \mathsf{c}_{\mathsf{sum,l}} & : & (op_1, op_2 : \mathsf{PosOp})(\mathsf{IndOp}\ op_1) \to (\mathsf{IndOp}\ (\mathsf{op\text{-}sum}\ op_1\ op_2)) \\ \quad \mathsf{c}_{\mathsf{sum,r}} & : & (op_1, op_2 : \mathsf{PosOp})(\mathsf{IndOp}\ op_2) \to (\mathsf{IndOp}\ (\mathsf{op\text{-}sum}\ op_1\ op_2)) \\ \quad \mathsf{c}_{\mathsf{fun}} & : & (K : *)(op : \mathsf{PosOp})(K \to (\mathsf{IndOp}\ op)) \to (\mathsf{IndOp}\ (\mathsf{op\text{-}fun}\ K\ op)) \\ \multicolumn{3}{l}{\mathsf{end.}} \end{array}$$

Lemma 1. *For every* op: PosOp*,* $X : * \vdash (\mathsf{IndOp}\ X\ op)$ *is a positive operator.*

Proof Just check that the requirements of the new clause in Definition 4 are satisfied. $\square$

Thus, in the type system of Coq, we can associate an inductive type to every element of PosOp:

$$\mathsf{IndType} := [op : \mathsf{PosOp}]\mu_X(\mathsf{IndOp}\ X\ op) : \mathsf{PosOp} \to *.$$

Whenever we have a family of type operators $X : * \vdash \Psi[X] : I \to *$ defined by recursion on an inductive type I, we can associate to it a function $f_\Psi : I \to \mathsf{PosOp}$

and obtain the family of inductive types as $[x\colon I](\mathsf{IndType}\ (f_\Psi\ x))$. For example, the family Ψ from Formula 3 is translated into the function

$$\begin{aligned} &f_\Psi\colon \mathbb{N} \to \mathsf{PosOp} \\ &\quad (f_\Psi \quad 0 \quad) := (\mathsf{op\text{-}const}\ \mathbb{N}_1) \\ &\quad (f_\Psi\ (S\ n)) := (\mathsf{op\text{-}sum}\ (\mathsf{op\text{-}const}\ \mathbb{N}_1)\ (\mathsf{op\text{-}prod}\ \mathsf{op\text{-}id}\ (f_\Psi\ n))) \end{aligned}$$

Moreover, we can avoid this translation by proving directly the positivity of the original operators inside type theory.

Definition 6. *The predicate* $\mathsf{Positive}\colon (* \to *) \to *$ *is inductively defined by the following rules:*

$$\frac{K\colon *}{(\mathsf{pos\text{-}const}\ K)\colon (\mathsf{Positive}\ [X\colon *]K)}$$

$$\frac{}{\mathsf{pos\text{-}id}\colon (\mathsf{Positive}\ [X\colon *]X)}$$

$$\frac{\Phi_1\colon * \to * \quad \Phi_2\colon * \to * \quad p_1\colon (\mathsf{Positive}\ \Phi_1) \quad p_2\colon (\mathsf{Positive}\ \Phi_2)}{(\mathsf{pos\text{-}prod}\ \Phi_1\ \Phi_2\ p_1\ p_2)\colon (\mathsf{Positive}\ [X\colon *](\Phi_1\ X) \times (\Phi_2\ X))}$$

$$\frac{\Phi_1\colon * \to * \quad \Phi_2\colon * \to * \quad p_1\colon (\mathsf{Positive}\ \Phi_1) \quad p_2\colon (\mathsf{Positive}\ \Phi_2)}{(\mathsf{pos\text{-}sum}\ \Phi_1\ \Phi_2\ p_1\ p_2)\colon (\mathsf{Positive}\ [X\colon *](\Phi_1\ X) + (\Phi_2\ X))}$$

$$\frac{K\colon * \quad \Phi\colon * \to * \quad p\colon (\mathsf{Positive}\ \Phi)}{(\mathsf{pos\text{-}fun}\ K\ \Phi\ p)\colon (\mathsf{Positive}\ [X\colon *]K \to (\Phi\ X))}$$

It is straightforward to define a function $\mathsf{pos\text{-}code}\colon (\Phi\colon * \to *)(\mathsf{Positive}\ \Phi) \to \mathsf{PosOp}$ by recursion on the proof of $(\mathsf{Positive}\ \Phi)$. In conclusion, given a recursive family of operators, we can prove by induction that every element of the family is positive and then obtain the recursive family of inductive types by composing $\mathsf{IndType}$ and $\mathsf{pos\text{-}code}$.

Lemma 2. *Every type operator* $\Phi\colon * \to *$ *such that* $(\mathsf{Positive}\ \Phi)$ *is provable, has an initial algebra.*

Finally we can apply this method to the strongly positive operators.

Theorem 4. *If* $X\colon * \vdash \Phi[X]\colon *$ *is a strongly positive operator, then there is a proof of* $(\mathsf{Positive}\ [X]\Phi[X])$.

Proof We just formalize the proof of Theorem 1. Since we are now developing the proof inside type theory, the requirement that no free variable except X appears in Φ is no longer necessary. Hence the result holds for every strongly positive operator. □

7 Conclusion

We have considered the problem of defining families of inductive types whose constructors are given by recursion. These families occur naturally in some developments of abstract mathematics in type theory. We characterized them with

the notion of strong positive operator. We described a model of them in type theory that uses wellorderings. We showed that a more manageable model can be constructed in a type theory with an extended notion of inductive definition. Finally we generalized the later model to a complete internalization of inductive definitions. This last part was completely formalized in Coq.

8 Acknowledgements

I am indebted to Henk Barendregt and Herman Geuvers for their help. The first discussed the content of this article with me and provided many illuminating comments and suggestions. The second read a first draft of the paper and helped me improve it. But if there are still mistakes in it, the responsibility is solely mine.

References

1. H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 117–309. Oxford University Press, 1992.
2. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjanin Werner. *The Coq Proof Assistant Reference Manual. Version 6.3.* INRIA, 1999.
3. Bruno Barras and Benjamin Werner. Coq in Coq. Draft paper, 2000.
4. G. Barthe, M. Ruys, and H. P. Barendregt. A two-level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs (TYPES'95)*, volume 1158 of *LNCS*, pages 16–35. Springer, 1995.
5. Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software. Third International Symposium, TACS'97*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.
6. Venanzio Capretta. Universal algebra in type theory. In Yves Bertot, Gilles Dowek, André Hirschowits, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *LNCS*, pages 131–148. Springer-Verlag, 1999.
7. Venanzio Capretta. Equational reasoning in type theory. `http://www.cs.kun.nl/~venanzio`, 2000.
8. Thierry Coquand. An analysis of Girard's paradox. In *Proceedings, Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.
9. Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf, editor, *Proceedings of Colog '88*, volume 417 of *LNCS*. Springer-Verlag, 1990.
10. Peter Dybjer. Representing Inductively Defined Sets by Wellorderings in Martin-Löf Type Theory. *Theoretical Computer Science*, 176:329–335, 1997.

11. Herman Geuvers. Inductive and coinductive types with iteration and recursion. In Bengt Nordström, Kent Pettersson, and Gordon Plotkin, editors, *Proccedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 193–217, 1992. `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.dvi.Z`.
12. Eduardo Giménez. A Tutorial on Recursive Types in Coq. Technical Report 0221, Unité de recherche INRIA Rocquencourt, May 1998.
13. Douglas J. Howe. Computational metatheory in Nuprl. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 238–257. Springer-Verlag, 1988.
14. Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.
15. Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.
16. Ralph Matthes. Monotone (co)inductive types and positive fixed-point types. *Theoretical Informatics and Applications*, 33:309–328, 1999.
17. Paul Francis Mendler. *Inductive Definition in Type Theory.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1987.
18. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Clarendon Press, 1990.
19. C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, 1993. LIP research report 92-49.
20. Holger Pfeifer and Harals Rueß. Polytypic proof construction. In Yves Bertot, Gilleds Dowek, André Hirschowits, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *LNCS*, pages 54–72. Springer-Verlag, 1999.
21. F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*. Springer-Verlag, 1990. technical report CMU-CS-89-209.
22. Mark Ruys. *Studies in Mechanical Verification of Mathematical Proofs.* PhD thesis, Computer Science Institute, University of Nijmegen, 1999.
23. Milena Stefanova. *Properties of Typing Systems.* PhD thesis, Computing Science Institute, University of Nijmegen, 1999.