Inductive types in constructive languages

Bruin, Peter Johan de

Chapter 5

Specifying inductive types

In the previous chapter, we introduced the notion of initial algebra. By viewing inductive types as initial algebras, we can define them up to isomorphism by giving the appropriate signature. However, not all algebra signatures have initial algebras. In this chapter we seek schemes for signatures that do have initial algebras, such that concrete inductive type definitions fit into the scheme.

We discuss only the abstract form that such schemes may take, not the concrete syntax that has to be defined in order to write signatures in a concrete language. We start with single inductive types, where an admissible functor F is specified by means of a family of sets. The generalization to mutually inductive types in section 5.2 diverges into several alternatives. In section 5.3 we have a look at producing admissible functors through inductive rules themselves. The use of positive type expressions may be seen as a special case of this.

The name '**Type**' as we use it stands for any universe $\mathbf{Type}_i$ from the hierarchy. But it is relevant that inductive types in one universe $\mathbf{Type}_i$ are understood to be initial algebras in all higher universes too, in order that one can use recursion to define other types.

5.1 Single inductive types

Suppose we want to introduce a single inductive type, $T\colon\mathbf{Type}$. In sections 4.3 and 4.4 we have seen how giving a signature consisting of a collection of constructors and equational axioms suffices to describe T. We discuss two ways to abstractly specify this collection.

We postpone the introduction of equations. So, in this section, we stipulate that objects built either by different constructors, or by one constructor from different arguments, are always different.

5.1.1 Operator domains

The most common approach in classical set theory (see for example Manes [54]) of giving a general scheme for initial algebras, is to identify for each constructor θ its arity as a

cardinal number $p\colon \mathbf{Card}$ so that $\theta\colon T^p \to T$. Note that a constructor with arguments of other types, say $\theta\colon A \times T^p \to T$, can be regarded as a family of constructors $\theta_{a:A}\colon T^p \to T$.

Subsequently, all constructors with equal arity p have to be collected in a single family τ_p indexed by a set Ω_p, so we get the typing

$$\tau_p\colon (T^p \to T)^{\Omega_p}$$

or equivalently

$$\tau_p\colon \Omega_p \times T^p \to T\ .$$

Thus, the family of sets

$$\Omega\colon \mathbf{Set}^{\mathbf{Card}}$$

determines the number and arity of all constructors. It is called an *operator domain*. The pair $(T;\tau)$ forms an F-algebra where:

$$F.X := \Sigma(p\colon \mathbf{Card} :: \Omega_p \times T^p)$$

Unfortunately, taking a functor of this form does not guarantee that an initial F-algebra exists in **SET**. We shall see that it does exist when Ω is *bounded*, i.e. Ω_n is empty for all n above some cardinal m. Also, when Ω_0 is empty, then the empty algebra, consisting of $T := \emptyset$, τ_0 the empty tuple, and all other τ_p being tuples of (empty) functions $t \mapsto t_0$, is trivially initial. But:

Theorem 5.1 *When Ω is not bounded and Ω_0 is nonempty, then Ω does not have an initial algebra in* **SET**.

Proof. Suppose $(T;\tau)$ were initial; we shall define (with choice) an injection $f\colon \mathbf{Set} \to T$ by means of set-recursion (recursion over the wellfounded relation $\in$, (A.7)).
For any set s, let p be its cardinality and choose a surjection $\phi_s\colon p \to s$. Let q be the least cardinal $q \geq p$ such that Ω_q is nonempty, and choose $e_s\colon \Omega_q$. Then define

$$f.s \ := \ \tau_q e_s.t \ \underline{\text{where}}\ t_{r:<q} := \begin{cases} f.(\phi_s.r) & \text{if } 0 \leq r < p \\ f.(\phi_s.0) & \text{if } p \leq r < q \end{cases}$$

Note that in the second case, $\phi_s.0$ is defined because if $q > p$ then Ω_p is empty so $p > 0$.
One checks easily that $f.s = f.s'$ implies $s = s'$. But such an injection cannot exist. ■

Objections against the use of operator domains in type theory are the following:

- As we just showed, extra conditions are needed to guarantee existence of an initial algebra with operator domain Ω.
- It may be unnatural or in constructive logic impossible to identify the arity as a cardinal number. To overcome this, one can use types rather than cardinals, so that $\Omega\colon \mathbf{Type}^{\mathbf{Type}}$.
- It may be unnatural to group constructors according to their arity. For example, an algebra may contain a family of constructors $\theta_{n:\omega}\colon T^n \to T$ which one would prefer to keep apart from other constructors for type T.

5.1.2 Operators with arity

An approach better fitted to type theory is to specify the arity of each constructor τ_j as a type. This is really the algebraic specification approach, but with finite sets replaced by possibly infinite types. The general form is:

$$\tau_{a:A}\colon T^{Ba} \to T \tag{5.1}$$

So, the signature is characterized by the type A, the index domain of the constructors, and the tuple of types B, Ba being the index domain of the constructor with index $a\colon A$. We call the pair $(A;B)\colon \mathsf{Fam}\,\mathbf{Type}$ an *operator specification*, and the corresponding signature is $(1, A; (X \mapsto (a :: X^{Ba})), (X \mapsto (a :: X)))$.

If we have an inductive type characterized by a list of constructors with arguments of other types as well, we can transform these into the form of (5.1) as follows. First, write the constructor types as:

$$\tau_{j:M}\colon \Sigma(x\colon A'_j :: T^{B'jx}) \to T \tag{5.2}$$

Next, the index j can be eliminated by transforming τ into (5.1) where we substitute

$$\begin{aligned} A\colon \mathbf{Type} &:= \Sigma(M; A') \\ B(j;x) &:= B'_j x \\ \tau_{(j;x)} &:= t \mapsto \tau_j.(x; (y :: t(j;y))) \end{aligned}$$

The types of τ in (5.2) and in (5.1) are isomorphic:

$$\begin{aligned} \Pi(j\colon M :: \Sigma(x\colon A'_j :: T^{B'jx}) \to T) &\cong \Pi(j\colon M :: \Pi(x\colon A'_j :: T^{B'jx} \to T)) \\ &\cong \Pi((j;x)\colon \Sigma(M; A') :: T^{B'jx} \to T) \end{aligned}$$

Example 5.1 In example 3.1 (natural numbers), we had two constructors, namely "zero" with 0 arguments, and "successor" with 1 argument. That is, $\tau_0\colon T^0 \to T$ and $\tau_1\colon T^1 \to T$, which results in the operator specification $(A := 2;\ B := (0,1))$.

For example 3.2 (lists over E), we have originally

$$\begin{aligned} &\tau_0\colon 1 \to T \\ &\tau_1\colon E \times T \to T \end{aligned}$$

which is first transformed into

$$\begin{aligned} &\tau_0\colon \Sigma(0\colon 1 :: T^0) \to T \\ &\tau_1\colon \Sigma(e\colon E :: T^1) \to T \end{aligned}$$

and next into (5.1) where $A := 1 + E$ and $B := ((0;0) :: 0 \mid (1;e) :: 1)$. We might use a labeled sum for A, for example:

$$A ::= \mathsf{empty} \mid \mathsf{cons}(E)\,;\ B := (\mathsf{empty} :: 0 \mid \mathsf{cons}(e) :: 1)$$

(End of example)

As a third step, one can replace the constructor family τ by a single constructor,

$$\tau\colon \Sigma(x\colon A :: T^{Bx}) \to T \ . \tag{5.3}$$

Note that we can define a functor $F\colon \mathbf{TYPE} \to \mathbf{TYPE}$ by:

$$\begin{array}{rcl} F.(U\colon \mathbf{Type}) & := & \Sigma(x\colon A :: U^{Bx}) \ , \\ F.(f\colon U \to V) & := & (x;t) \mapsto (x;(y :: f.t_y)) \ . \end{array}$$

We will call a functor *polynomial* iff it is (naturally) isomorphic with a functor of this form, as it is a sum of products. It is well known that for polynomial F, an initial F-algebra does always exist, and we will name it μF. Two different constructions of μF are given in chapter 8. A final F-coalgebra exists too and is named νF.

The type inductively defined by F is the carrier of the algebra μF, which is called μF too. This type is a fixed point of functor F, modulo isomorphism. Not all functors do have fixed points; for example the powerset functor (with $\mathcal{P}.f := X \mapsto \{x\colon\in X :: f.x\,\}$) cannot have one for cardinality reasons. But most type-construction principles, such as generalized sum and product, and taking fixed points itself, transform polynomial functors into polynomial functors. This is exploited in the next subsection. Also, for many theoretical purposes, it's simpler to deal with a functor than with A and B explicitly.

An operator specification that corresponds to an operator domain Ω is

$$(A := \Sigma(\mathbf{Card}; \Omega);\ B_{(n;a):A} := n) \ .$$

Note that $\Sigma(\mathbf{Card}; \Omega)$ is a set indeed if and only if Ω is bounded.

Conversely, given an operator specification $(A; B)$, a corresponding operator domain is

$$\Omega_p := \{\, a\colon A \mid\colon B_a \cong p\} \ .$$

This assumes that every type is isomorphic to some cardinal p, which requires the axiom of choice.

5.1.3 The wellordering of a single inductive type

The inductive type T characterized by operator specification $(A; B)$ is wellordered by the subterm relation $|<| := |\prec|^{(+)}$, where $\prec$ is the immediate subterm relation:

$$|\prec| \ := \ \{x\colon A;\ t\colon T^{Bx};\ y\colon Bx :: (t_y, \tau.(x;t))\,\} \ . \tag{5.4}$$

Type T is actually Martin-Löf's so-called wellordering type $\mathsf{W}(A, B)$ described in [56] where the constructor is called sup after *supremum* as, for $a\colon A$ and $t\colon \mathsf{W}(A, B)^{Ba}$, $\mathsf{sup}(a; t)\colon \mathsf{W}(A, B)$ is the supremum of the family $(y\colon Ba :: t_y)$ with respect to $<$.

5.2 Mutually inductive types

By *mutually inductive types* we mean a family of types $T_{i:N}$ for some index domain N, that are inductively generated by constructors τ whose arguments may come from any

T_i from the family. The cardinality of N is normally greater than 1. Algebra $(T;\tau)$ can again be viewed as an initial $(N,M;F,G)$-algebra.

A well-founded relation over a family of types is simply a well-founded relation over the sum type of the family, $\Sigma(N;T)$. Specifying a family of N types by simultaneous induction is somewhat more complicated. We give two alternatives for the abstract specification of mutually inductive types. The first is the generalization of the polynomial functor approach of 5.1.2 to exponential categories, but the required generalization of "polynomial" is not evident. The second alternative is the abstract rendering of plain (multi-sorted) algebra signatures, mentioned in 4.7.

5.2.1 Using an exponential category

We may use an endo-functor on the exponential category $\mathbf{TYPE}^N$, i.e. $F\colon \mathbf{TYPE}^N \to \mathbf{TYPE}^N$, that is polynomial as defined below. Then the tuple of mutually inductive types and their constructors appears as an initial algebra $\mu F = (T\colon \mathbf{Type}^N;\ \tau\colon F.T \to T)$. As arrows in an exponential category are tuples of arrows in the component categories, τ consists of functions $\tau_i\colon (F.T)_i \to T_i$ for each $i\colon N$.

We give two ways to characterize such polynomial functors F on $\mathbf{TYPE}^N$. In both ways, a type A_i indexes the constructors for type T_i.

The first way, perhaps the most straightforward one, lets type $B((i;a),j)$ (also written $Biaj$) index the arguments of type T_j that the constructor for type T_i indexed by $a\colon A_i$ shall get. Thus, for some

$$A\colon \mathbf{Type}^N;\ B\colon \mathbf{Type}^{\Sigma(N;A)\times N}\ ,$$

we let the constructors be typed by

$$\tau_{i:N;a:A_i}:\ \ \Pi(j\colon N :: T_j^{Biaj}) \to T_i\ . \tag{5.5}$$

Equivalently, $\tau\colon F.T \to T$ where F is given by

$$(F.X)_i\ :=\ \Sigma(a\colon A_i :: \Pi(j\colon N :: X_j^{Biaj}))\ .$$

With the second way, $B_i a\colon \mathbf{Type}$ indexes all arguments that the constructor τ_{ia} shall get, and $si(a;k)\colon N$ (or $siak$) indicates what the type of the argument indexed by $k\colon B_i a$ must be. Thus, for some

$$A\colon \mathbf{Type}^N;\ B\colon \Pi(i\colon N :: \mathbf{Type}^{Ai});\ s\colon \Pi(i\colon N :: N^{\Sigma(Ai;Bi)})\ ,$$

we let the constructors be typed by

$$\tau_{i:N;a:Ai}:\ \ \Pi(k\colon B_i a :: T_{(siak)}) \to T_i\ . \tag{5.6}$$

Equivalently, $\tau\colon F.T \to T$ where F is given by

$$(F.X)_i\ :=\ \Sigma(a\colon A_i :: \Pi(k\colon B_i x :: X_{(siak)}))\ .$$

This is the approach proposed by Petersson [72, section 4]. An advantage over the plain algebra approach below is that one gets a simpler case-distinction construct when no recursion is needed.

A context-free grammar corresponds to an algebra specification of form (5.6), where all A_i and $B_i a$ are finite:

Example 5.2 The rose trees and forests of example 3.5 can be described by a context-free grammar, given a nonterminal E:

$$\begin{array}{lcl} \mathit{RTree} & ::= & E \surd \mathit{Forest} \\ \mathit{Forest} & ::= & \Box \mid \mathit{RTree} +\!\!< \mathit{Forest} \end{array}$$

This corresponds to an algebra (5.6), where

$$\begin{array}{rcl} N & := & 2; \\ A & := & (E, 2); \\ B & := & ((e :: 1),\ (0, 2)\,); \\ s & := & (((e; 0) :: 1),\ ((1; 0) :: 0 \mid (1; 1) :: 1)\,) \end{array}$$

Theorem 5.2 *Both characterizations* (5.5) *and* (5.6) *are reducible to each other, provided one has equality types.*

Proof. (5.6) $\Rightarrow$ (5.5): We can find an initial solution to (5.5) for some given $(A; B)$ by finding an initial solution to (5.6) with

$$B_i a := \Sigma(j{:}\, N :: Biaj);\ sia(j; k) := j\ ,$$

and then taking $\tau_{ia} := p \mapsto \tau_{ia}.((j; k) :: pjk)$.

(5.5) $\Rightarrow$ (5.6): Using equality, apply (5.5) with $Biaj := \{k{:}\, B_i a \mid{:}\, siaj = k\}$.

5.2.2 Plain algebra signatures

The choice above, that τ be an arrow in the category $\mathbf{TYPE}^N$, implied that each type T_i had its own (family of) constructor(s). Alternatively, one can give a single family of constructors and indicate for each one its codomain. This makes us return to the notion of *plain algebra* as we used in connection with Algebraic Specification in section 4.7, but here infinite families of constructors and arguments are allowed, rather than finite sequences. So given

$$M{:}\,\mathbf{Type};\ B{:}\,\mathbf{Type}^M;\ d{:}\,N^{\Sigma(M;B)};\ c{:}\,N^M\ ,$$

let τ be typed by

$$\tau_{j:M}{:}\,\Pi(k{:}\, B_j :: T_{(djk)}) \to T_{(cj)}\ . \tag{5.7}$$

So $(T; \tau)$ is not an F-algebra, but it is an initial $(N, M; F, G)$-algebra where

$$\begin{array}{rcl} (F.X)_j & := & \Pi(k{:}\, B_j :: X_{(djk)}) \\ (G.X)_j & := & X_{(cj)} \end{array}$$

Theorem 5.3 *Formulations* (5.6) *and* (5.7) *reduce to each other, provided one has equality types.*

Proof. (5.7) ⇒ (5.6): Apply (5.7) with $M := \Sigma(N; A)$; $B := B$; $d(i; a)k := siak$; $c(i; a) := i$.

(5.6) ⇒ (5.7): Here one needs equality. Apply (5.6) with $A_i := \{\, j : M \mid : cm =_N i\}$; $Bij := B_j$; $dijk := djk$. ■

This formulation is particularly suited to build types of proof trees for inductively defined relations:

Example 5.3 The type $P(n, m)$ of proof trees for $n < m$ as defined in example 3.7, together with appropriate constructors, is initial in the category of algebras

$$(P : \mathbf{Type}^{\mathbb{N}^2};\ \tau_{(0;n)} : 1 \to P(n, \mathsf{s}\, n),\ \tau_{(1;\, n,m)} : P(n, m) \to P(n, \mathsf{s}\, m))\ .$$

So here we have (5.7) with:

$$\begin{array}{rcl} N & := & \mathbb{N}^2 \\ M & := & \mathbb{N} + \mathbb{N}^2 \\ B & := & (0; n) :: 0 \mid (1;\ n, m) :: 1 \\ d & := & (1;\ n, m;\ 0) :: (n, m) \\ c & := & (0; n) :: (n, \mathsf{s}\, n) \mid (1;\ n, m) :: (n, \mathsf{s}\, m)\ . \end{array}$$

(End of example)

This example gives us an alternative way to define $(<)$: first define $(P; \tau)$ to be an initial algebra in the above category, and then define $n < m$ to hold iff $P(n, m)$ is nonempty. If one's calculus has inductive type definitions as a primitive, then this avoids the second order quantification occurring in example 3.7. Or, inductive relation definitions can be allowed as primitive themselves by stating that the category of predicates $T : \mathbf{PROP}^N$ with constructors typed by (5.7) has an initial object.

A drawback of this plain algebra approach is that it seems less suited for dualization; see paragraph 7.1.2.

5.3 Production rules for polynomial functors

Instead of requiring that the typings of operations have exactly a polynomial form like (5.1) or (5.3), the class of polynomial type expressions may be defined by production rules. This is based on the fact that the class of polynomial functors contains projections and constant functors, and is closed under taking sums, products and inductive types.

Let $\mathsf{PF}(N, M) : \subseteq (\mathbf{TYPE}^N \to \mathbf{TYPE}^M)$ be the subtype of polynomial functors as defined in subsection 5.2.1 and closed under isomorphism; $\mathsf{PF}(N)$ is $\mathsf{PF}(N, 1)$; note that $\mathsf{PF}(N, M) \cong \mathsf{PF}(N)^M$. We state the following fact:

Theorem 5.4 *Type* $\mathsf{PF}(N)$ *is closed under the following rules:*

$$i : N \ \vdash\ (X \mapsto X_i) \in \mathsf{PF}(N) \tag{5.8}$$

$$T : \mathbf{Type} \ \vdash\ (X \mapsto T) \in \mathsf{PF}(N) \tag{5.9}$$

$$F : \mathsf{Fam}\, \mathsf{PF}(N) \ \vdash\ \Sigma F \in \mathsf{PF}(N),\ \Pi F \in \mathsf{PF}(N) \tag{5.10}$$

$$F : \mathsf{PF}(N + M, M) \ \vdash\ (X \mapsto \mu(Y : \mathbf{Type}^M \mapsto F.(X, Y))) \in \mathsf{PF}(N, M), \tag{5.11}$$

$$(X \mapsto \nu(Y : \mathbf{Type}^M \mapsto F.(X, Y))) \in \mathsf{PF}(N, M) \tag{5.12}$$

where ΣF *abbreviates the sum functor* $(X \mapsto \Sigma(i\colon \mathsf{Dom}\, F :: F_i.X))$*, and similarly for* ΠF*.*

To understand (5.11), let $F\colon \mathsf{PF}(M+N, M)$, and note that for $X\colon \mathbf{Type}^N$ we have that $(Y \mapsto F.(X,Y))$ is an endofunctor in $\mathbf{TYPE}^M$ indeed, so $\mu(Y \mapsto F.(X,Y))$ is an M-tuple of types. We write this tuple as $\mu F.X$. To see that $\mu F\colon \mathbf{Type}^N \to \mathbf{Type}^M$ is a functor, let $g\colon X \to X'$ be an arrow in $\mathbf{TYPE}^N$; we have to find an arrow $\mu F.g$. We take

$$\mu F.g\colon \mu F.X \to \mu F.X' \;:=\; (\!|\mu F.X'; \phi|\!)$$

where

$$\phi\colon F.(X, \mu F.X') \to \mu F.X' \;:=\; F.(g, \mathsf{I}) \mathbin{\bar{\circ}} \tau \;.$$

Proving that each $(\mu F.X)_j$, for $j\colon M$, is polynomial in X is quite complicated; we do not try it here.

5.3.1 Positive type expressions

One possibility for employing the above observation in a language design is to syntactically distinguish polynomial functors as type expressions E_X that are (strictly) *positive* in X. We say that a type expression is positive in X iff all free occurrences of type variables X are positive, i.e. not within the domain of any product or function type, nor in the parameter or argument of any user-defined operation.

This is done for example in Nuprl [18], where one can form the inductive type $\mu(X \mapsto E_X)$, and in Paulin-Mohring's extension [22, 68] of CC, where one can form the inductive type that is closed under a finite number of constructors with positive argument types.

A drawback is that in building E_X from X, one can neither apply user-defined type constructors (as we did in our example 3.5, where a user-defined type of lists was applied in defining the type of rose trees), nor apply other constants or variables to X. (Normally one can replace the constant with its definition.)

5.3.2 A type of polynomial functors

As an alternative, one might include the class of polynomial functors as a primitive type itself, say $\mathsf{PF}(N, M)$, which is closed under composition and the operations listed in theorem 5.4. This would allow user definitions of polynomial functors, and polymorphic operations to be instantiated to polynomial functors. One has to find easy notations, for example noting the projection functor $(X \mapsto X_i)$ by the name of parameter i.

5.4 Adding equations

To any of the three approaches above, one can add syntactic or semantic equations as in section 4.4. Semantic equations can be used if one's language admits such a semantic approach, otherwise one can develop concrete syntax for syntactic terms and equations.

5.5 Conclusion

We characterized the signatures that may be admitted for defining inductive types, in the following ways.

1. Using a bounded operator domain–this approach does not fit very well to type theory
2. Giving an arity for each constructor, by which means one defines a polynomial functor
3. For mutually inductive types, generalizing the notion of polynomial functor to an exponential category in either of two ways
4. Using generalized plain algebra signatures, which leaves a bit more freedom for the type definition
5. Using polynomial functors characterized by production rules, or by syntactic conditions on type expressions