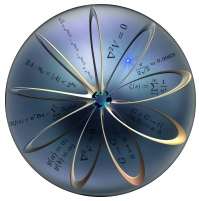

planetmath.org

Math for the people, by the people.

5.7 Generalizations of inductive types

Canonical name	57GeneralizationsOfInductiveTypes
Date of creation	2013-11-18 3:09:23
Last modified on	2013-11-18 3:09:23
Owner	PMBookProject (1000683)
Last modified by	rspuzio (6075)
Numerical id	2
Author	PMBookProject (6075)
Entry type	Feature
Classification	msc 03B15

The notion of inductive type has been studied in type theory for many years, and admits of many, many generalizations: inductive type families, mutual inductive types, inductive-inductive types, inductive-recursive types, etc. In this section we give an overview of some of these, a few of which will be used later in the book. (In `http://planetmath.org/node/87579`Chapter 6 we will study in more depth a very different generalization of inductive types, which is particular to *homotopy* type theory.)

Most of these generalizations involve allowing ourselves to define more than one type by induction at the same time. One very simple example of this, which we have already seen, is the coproduct $A + B$. It would be tedious indeed if we had to write down separate inductive definitions for $\mathbb{N} + \mathbb{N}$, for $\mathbb{N} + \mathbf{2}$, for $\mathbf{2} + \mathbf{2}$, and so on every time we wanted to consider the coproduct of two types. Instead, we make one definition in which A and B are variables standing for types; in type theory they are called **parameters**. Thus technically speaking, what results from the definition is not a single type, but a family of types $+ : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$, taking two types as input and producing their coproduct. Similarly, the type $\mathsf{List}(A)$ of lists is a family $\mathsf{List}(\_) : \mathcal{U} \to \mathcal{U}$ in which the type A is a parameter.

In mathematics, this sort of thing is so obvious as to not be worth mentioning, but we bring it up in order to contrast it with the next example. Note that each type $A + B$ is *independently* defined inductively, as is each type $\mathsf{List}(A)$. By contrast, we might also consider defining a whole type family $B : A \to \mathcal{U}$ by induction *together*. The difference is that now the constructors may change the index $a : A$, and as a consequence we cannot say that the individual types $B(a)$ are inductively defined, only that the entire family is inductively defined.

The standard example is the type of *lists of specified length*, traditionally called **vectors**. We fix a parameter type A, and define a type family $\mathsf{Vec}_n(A)$, for $n : \mathbb{N}$, generated by the following constructors:

- a vector $\mathsf{nil} : \mathsf{Vec}_0(A)$ of length zero,
- a function $\mathsf{cons} : \prod_{(n:\mathbb{N})} A \to \mathsf{Vec}_n(A) \to \mathsf{Vec}_{\mathsf{succ}(n)}(A)$.

In contrast to lists, vectors (with elements from a fixed type A) form a family of types indexed by their length. While A is a parameter, we say that $n : \mathbb{N}$ is an **index** of the inductive family. An individual type such as $\mathsf{Vec}_3(A)$ is not inductively defined: the constructors which build elements of $\mathsf{Vec}_3(A)$

take input from a different type in the family, such as $\mathsf{cons} : A \to \mathsf{Vec}_2(A) \to \mathsf{Vec}_3(A)$.

In particular, the induction principle must refer to the entire type family as well; thus the hypotheses and the conclusion must quantify over the indices appropriately. In the case of vectors, the induction principle states that given a type family $C : \prod_{(n:\mathbb{N})} \mathsf{Vec}_n(A) \to \mathcal{U}$, together with

- an element $c_{\mathsf{nil}} : C(0, \mathsf{nil})$, and
- a function $c_{\mathsf{cons}} : \prod_{(n:\mathbb{N})} \prod_{(a:A)} \prod_{(\ell:\mathsf{Vec}_n(A))} C(n, \ell) \to C(\mathsf{succ}(n), \mathsf{cons}(a, \ell))$

there exists a function $f : \prod_{(n:\mathbb{N})} \prod_{(\ell:\mathsf{Vec}_n(A))} C(n, \ell)$ such that

$$f(0, \mathsf{nil}) \equiv c_{\mathsf{nil}}$$
$$f(\mathsf{succ}(n), \mathsf{cons}(a, \ell)) \equiv c_{\mathsf{cons}}(n, a, \ell, f(\ell)).$$

One use of inductive families is to define *predicates* inductively. For instance, we might define the predicate $\mathsf{iseven} : \mathbb{N} \to \mathcal{U}$ as an inductive family indexed by $\mathbb{N}$, with the following constructors:

- an element $\mathsf{even}_0 : \mathsf{iseven}(0)$,
- a function $\mathsf{even}_{ss} : \prod_{(n:\mathbb{N})} \mathsf{iseven}(n) \to \mathsf{iseven}(\mathsf{succ}(\mathsf{succ}(n)))$.

In other words, we stipulate that 0 is even, and that if n is even then so is $\mathsf{succ}(\mathsf{succ}(n))$. These constructors "obviously" give no way to construct an element of, say, $\mathsf{iseven}(1)$, and since iseven is supposed to be freely generated by these constructors, there must be no such element. (Actually proving that $\neg\mathsf{iseven}(1)$ is not entirely trivial, however). The induction principle for iseven says that to prove something about all even natural numbers, it suffices to prove it for 0 and verify that it is preserved by adding two.

Inductively defined predicates are much used in computer formalization of mathematics and software verification. But we will not have much use for them, with a couple of exceptions in `http://planetmath.org/103ordinalnumbers`§10.3,`http://pl`

Another important special case is when the indexing type of an inductive family is finite. In this case, we can equivalently express the inductive definition as a finite collection of types defined by *mutual induction*. For instance, we might define the types even and odd of even and odd natural numbers by mutual induction, where even is generated by constructors

- $0 : \mathsf{even}$ and
- $\mathsf{esucc} : \mathsf{odd} \to \mathsf{even}$,

while odd is generated by the one constructor

- $\mathsf{osucc} : \mathsf{even} \to \mathsf{odd}$.

Note that even and odd are simple types (not type families), but their constructors can refer to each other. If we expressed this definition as an inductive type family $\mathsf{paritynat} : \mathbf{2} \to \mathcal{U}$, with $\mathsf{paritynat}(0_{\mathbf{2}})$ and $\mathsf{paritynat}(1_{\mathbf{2}})$ representing even and odd respectively, it would instead have constructors:

- $0 : \mathsf{paritynat}(0_{\mathbf{2}})$,
- $\mathsf{esucc} : \mathsf{paritynat}(0_{\mathbf{2}}) \to \mathsf{paritynat}(1_{\mathbf{2}})$,
- $\mathsf{oesucc} : \mathsf{paritynat}(1_{\mathbf{2}}) \to \mathsf{paritynat}(0_{\mathbf{2}})$.

When expressed explicitly as a mutual inductive definition, the induction principle for even and odd says that given $C : \mathsf{even} \to \mathcal{U}$ and $D : \mathsf{odd} \to \mathcal{U}$, along with

- $c_0 : C(0)$,
- $c_s : \prod_{(n:\mathsf{odd})} D(n) \to C(\mathsf{esucc}(n))$,
- $d_s : \prod_{(n:\mathsf{even})} C(n) \to D(\mathsf{osucc}(n))$,

there exist $f : \prod_{(n:\mathsf{even})} C(n)$ and $g : \prod_{(n:\mathsf{odd})} D(n)$ such that

$$\begin{aligned} f(0) &\equiv c_0 \\ f(\mathsf{esucc}(n)) &\equiv c_s(g(n)) \\ g(\mathsf{osucc}(n)) &\equiv d_s(f(n)). \end{aligned}$$

In particular, just as we can only induct over an inductive family "all at once", we have to induct on even and odd simultaneously. We will not have much use for mutual inductive definitions in this book either.

A further, more radical, generalization is to allow definition of a type family $B : A \to \mathcal{U}$ in which not only the types $B(a)$, but the type A itself, is defined as part of one big induction. In other words, not only do we specify constructors for the $B(a)$s which can take inputs from other $B(a')$s, as with

inductive families, we also at the same time specify constructors for A itself, which can take inputs from the $B(a)$s. This can be regarded as an inductive family in which the indices are inductively defined simultaneously with the indexed types, or as a mutual inductive definition in which one of the types can depend on the other. More complicated dependency structures are also possible. In general, these are called **inductive-inductive definitions**. For the most part, we will not use them in this book, but their higher variant (see `http://planetmath.org/node/87579`Chapter 6) will appear in a couple of experimental examples in `http://planetmath.org/node/87585`Chapter 11.

The last generalization we wish to mention is **inductive-recursive definitions**, in which a type is defined inductively at the same time as a *recursive* function on it. That is, we fix a known type P, and give constructors for an inductive type A and at the same time define a function $f : A \to P$ using the recursion principle for A resulting from its constructors — with the twist that the constructors of A are allowed to refer also to the values of f. We do not yet know how to justify such definitions from a homotopical perspective, and we will not use any of them in this book.