Excerpt from The Structure of Nuprl's Type theory, Constable
(see www.nuprl.org under Publications)

2.6 Inductive Types

Defining types in terms of themselves is quite common in programming, often pointers are used in the definition (in Pascal and C for example), but in languages like ML, direct recursive definitions are possible. For example, a list of numbers, L can be introduced by a definition like

$$\textit{define type} \qquad L = \mathbb{N} + (\mathbb{N} \times L).$$

In due course, we will give conditions telling when such definitions are sensible, but for now let us understand how elements of such a type are created. Basically a type of this kind will be sensible just when we understand exactly what elements belong to the type. Clearly elements like $inl(0), inl(1), \ldots$, are elements. Given them it is clear that $inr(\langle 0, inl(0)\rangle), inr(\langle 1, inl(0)\rangle)$ and generally $inr(\langle n, inl(m)\rangle)$ are elements, and given these, we can also build

$$inr(\langle k, inr(\langle n, inl(m)\rangle)\rangle) \text{ and so forth.}$$

In general, we can build elements in any of the types

$$\begin{array}{l} \mathbb{N} + \mathbb{N} \times Y \\ \mathbb{N} + \mathbb{N} \times (\mathbb{N} + \mathbb{N} \times Y) \\ \mathbb{N} + \mathbb{N} \times (\mathbb{N} + \mathbb{N} \times (\mathbb{N} + \mathbb{N} \times Y)) \end{array}$$

The key question about this process is whether we want to allow *anything else* in the type L. Our decision is **no**, we want only elements obtained by this finite process.

In set theory, we can understand similar inductive definitions, say a *set* L such that $L = \mathbb{N} \cup (\mathbb{N} \times L)$, as least fixed points of monotonic operators $F : Set \to Set$. In general, given such an operator F, we say that the set inductively defined by F is the least *F–closed* set, call it $I(F)$. We define it as

$$I(F) = \cap\{Y | F(Y) \subseteq Y\}.$$

We use set theory as a *guide* to justify recursive type definitions.

For the sake of defining monotonicity, we use the subtyping relation, $S \subseteq T$. This holds just when the elements of S are also elements of T, and the equality on S and T is the same. For example,

$$\text{if} \qquad S \subseteq T \qquad \text{then} \qquad \mathbb{N} + \mathbb{N} \times S \;\subseteq\; \mathbb{N} + \mathbb{N} \times T.$$

Def. Given a type expression $F : \mathit{Type} \to \mathit{Type}$ such that if $T_1 \subseteq T_2$ then $F(T_1) \subseteq F(T_2)$, then write $\mu X.F(X)$ as the *type inductively defined by F.*

To construct elements of $\mu X.F(X)$, we basically just *unwind* the definition. That is,

$$\boxed{\text{if} \quad t \in F(\mu X.F(X)) \qquad \text{then} \quad t \in \mu X.F(X).}$$

We say that $t_1 = t_2$ in $\mu X.F(X)$ iff $t_1 = t_2$ in $F(\mu X.F(X))$.

The power of recursive types comes from the fact that we can define total computable functions over them very elegantly, and we can prove properties of elements recursively. Recursive definitions are given by this term,

$$\boxed{\mu\text{–}ind(a; f, z.b)}$$

called a *recursor* or *recursive-form.* It obeys the computation rule

$$\boxed{\begin{array}{l}\mu\text{–}ind(a; f, z.b) \quad \text{evaluates in one step to} \\ b[a/z, (y \mapsto \mu\text{–}ind(y; f, z.b))/f].\end{array}}$$

(Note in this rule we use the notation $b[s/x, t/y]$ to mean that we substitute s for x and t for y in b.)

typing

The way we figure out a type for this form is given by a simple rule. We say that

$$\mu\text{–}ind(a; f, z.b) \qquad \text{is in type } B$$

provided that $a \in \mu X.F(X)$, and if when Y is a Type, and $Y \subseteq \mu X.F(X)$, and z belongs to $F(Y)$, and f maps Y to B, then b is of type B.

induction

The principle of inductive reasoning over $\mu X.F(X)$ is just this.

μ–induction

Let $R = \mu X.F(X)$ and assume that Y is a subtype of R and that for all x in Y, $P(x)$ is true. (This is the induction hypothesis.) Then if we can show $\forall z : F(Y).P(z)$, we can conclude

$$\forall x : R.P(x).$$

With this principle and the form *μ–ind*, we can write programs over recursive types and prove properties of them. The approach presented here is quite abstract, so it applies to a large variety of specific programming languages. It also stands on its own as a mathematical theory of types.

typing rules

We will write these informal rules as inference rules. To connect to the Nuprl account I will use its notation which is $rec(X.T)$ for $\mu X.T$ and $rec\_ind(a; f, z.b)$ for $\mu\_ind(a; f, z.b)$. In our Core Type Theory, recursive types are written as $\mu X.T$ where μ stands for the least fixed point operator. This is perhaps more standard notation; however, $rec(X.T)$ is the Nuprl syntax and is mnemonic.

Here is the rule which says $rec(X.T)$ is a type using a sequent style presentation of rules:

$$\frac{E, X : Type \vdash T \in Type}{E \vdash rec(X.T) \in Type}$$

note T monotone

For example, we can derive $rec(N.\mathbf{1} + \mathbf{1} \times N)$ is a type as follows:

$$\frac{\vdash \mathbf{1} \in Type \qquad \dfrac{\vdash \mathbf{1} \in Type \quad E, N : Type \vdash N \in Type}{E \vdash \mathbf{1} \times N \in Type}}{\dfrac{E, N : Type \vdash (\mathbf{1} + \mathbf{1} \times N) \in Type}{E \vdash rec(N.\mathbf{1} + \mathbf{1} \times N) \in Type}}$$

Remember that $\mathbf{1}$ is a primitive type. This type is essentially "$\mathbf{1}$ list."

To introduce elements we use the following rule of unrolling:

$$\frac{E \vdash t \in T[rec(X.T)/X]}{E \vdash t \in rec(X.T)}$$

Here are some examples on in $rec(N.1 + 1 \times N)$. Recall that type $1 = \{\bullet\}$.

The nil list is derived as:

$$\frac{\dfrac{\vdash \bullet \in 1}{N : Type \vdash inl(\bullet) \in 1 + rec(N.1 + 1 \times N)}}{\vdash inl(\bullet) \in rec(N.1 + 1 \times N)}$$

Lists are derived as:

$$\frac{\dfrac{\dfrac{\vdash \bullet \in 1 \quad \vdash inl(\bullet) \in rec(N.1 + 1 \times N)}{\vdash pair(\bullet; inl(\bullet)) \in 1 \times rec(N.1 + 1 \times N)}}{\vdash inr(pair(\bullet; inl(\bullet))) \in 1 + 1 \times rec(N.1 + 1 \times N)}}{\vdash inr(pair(\bullet; inl(\bullet))) \in rec(N.1 + 1 \times N)}$$

To make a decision about, for example, whether l is nil or not, unroll on the left:

$$\frac{E, l : 1 + 1 \times rec(N.1 + 1 \times N) \vdash g \in G}{E, l : rec(N.1 + 1 \times N) \vdash g \in G}$$

recursion combinator typing

This version of recursive types allows "primitive recursion" over recursive types also called the *natural recursion*. The Nuprl syntax is

$$rec\_ind(t; f, z.g)$$

The usual convention for the syntax also applies here: t and g are two sub-terms and g is in the scope of two binding variables f and z. (We define the recursion combinator to be $\lambda(x.rec\_ind(x; f, z.g))$.)

For evaluation, we recall the computation rule:

$$rec\_ind(t; f, z.g) \rightarrow_\rho g[t/z, \lambda(w.rec\_ind(w; f, z.g/f))]$$

called ρ reduction. We don't evaluate t, but simply substitute it into g.

Typing of the recursion combinator is the key part:

$$\frac{E \vdash t \in rec(X.T) \quad E, X : Type, z : T, f : X \rightarrow G \vdash g \in G}{E \vdash rec\_ind(t; f, z.g) \in G}$$

Where is the induction? It's in the top right-hand side. For sanity check, unwind: $rec\_ind(t; f, z.g) \in G$ reduces to $g[t/z, \lambda(w.rec\_ind(w; f, z.g/f))] \in G$. As we expect, $g \in G$, z and y have the same type, f is a function from elements of recursive type to G. Then we bottom out on X. Thus X in $f : X \rightarrow G$ is the key part.

The longer you know this rule, the clearer it gets. It summarizes recursion in one rule.

definition of *ind* using *rec_ind*

Here is how to define a natural recursion combinator on $\mathbb{N}$, call it $ind(n; b; u, i.g)$, using $rec\_ind()$. Let's say how this behaves. The base case is:

$$ind(0; b; u, i.g) \to b$$

The inductive case is:

$$ind(succ(n); b; u, i.g) \to g[n/u, ind(n; b; u, i.g)/i]$$

We combine the base and the inductive cases into one rule as follows:

$$\frac{E \vdash n \in N \quad u : N, i : G[u/x] \vdash g \in G[succ(u)/x]}{E \vdash ind(n; b; u, i.g) \in G[n/x]}$$

To derive this rule for *ind* we can encode $\mathbb{N}$ into **1** list:

1. $\mathbf{1} = \{\bullet\}$
2. $0 = inl(\bullet)$
3. We can define the successor of m as $succ(m) = inr\langle \bullet, m\rangle$
4. $\mathbf{N}$ is isomorphic to $rec(N.\mathbf{1} + \mathbf{1} \times N)$

This presentation of N doesn't have any intrinsic value; it's just a way to look at N (for an alternative presentation try deriving *ind* from $rec(N.\ \mathbf{1}\ + N)$). It gives us an existence proof for the class of natural numbers.

Given the above setup, we want to derive the rule for `ind`:

$$\frac{E \vdash n \in \mathbb{N} \quad E \vdash b \in G[0/x] \quad E,\ u : \mathbb{N},\ i : G[u/x] \vdash g \in G[succ(u)/x]}{E \vdash ind(n; b; u, i.g) \in G[n/x]}$$

using the rule of *rec_ind*:

$$\frac{E \vdash t \in rec(X.T) \quad E,\ X : \text{Type},\ x : T,\ f : (y : X \to G[y/x]) \vdash g \in G}{E \vdash rec\_ind(t; f, x.g) \in G[t/x]}$$

Notice the following facts in the above two rules:

- In the rule for induction the first hypothesis denotes that we are doing induction over the natural nymbers, the second one is the base case of this induction (notice that we substitute x by zero) and the third is the induction hypothesis, by which we prove the fact for the successor of u assuming it for u.
- In the rule for recursive-induction x should be of type T (which is the body of the recursive type). This is because we want to be able to compute the predecessors of x.

If we want to indicate the new variables we use in the recursive induction rule we write the rule adding this information:

$$\frac{E \vdash t \in rec(X.T) \quad E,\ X : \text{Type},\ x : T,\ f : (y : X \rightarrow G[y/x]) \vdash g \in G}{E \vdash rec\_ind(t; f, x.g) \in G[t/x]}$$

This is an elegant induction form describing any possible recursive definition.

Lets see how this works in the case of building the *ind* rule. We constuct a $\hat{g}$ from g, so that if we use $\hat{g}$ in *rec_ind* we will derive $ind(n; b; u, i.g)$. Before we present $\hat{g}$, lets try to get a bit of the intuition. The most important part of the *ind* rule is its induction hypothesis. We want to specialise the *rec_ind* rule so that we can pull out and isolate this induction hypothesis.

The *idea* is the following:

> Let i be the function f applied to u, $i = f(u)$, where u is the predecessor of x we are looking at. We can get this predecessor u (as we have already noticed before) by using the hypothesis $x : T$ and by decomposing x.

For example, if T is the type of **1** list, then $T = \mathbf{1} + \mathbf{1} \times N$. We do a case split and we get two cases; in the first case we get **1** which represents zero, and in the second case we get $\mathbf{1} \times N$ which is the successor. We can construct $\hat{g}$, given g of the `ind` form, by taking a close look at the structure of x:

- T is a disjunct (and, hence, $\hat{g}$ is a *decide*),
- T's *inl* side has the zero, so we need it as the base case (which is b) and
- T's *inr* side is a pair which we want to spread and use. In the spread, u will denote the predecessor of x.

It seems that the following form for $\hat{g}$ can give us all we need:

$$\hat{g} = decide(x; z.b; p.spread(p; \bullet, u.g[f(u)/i]))$$

As an **exercise** you can prove that $\hat{g} \in G$ (*hint*: use the fact that $b \in G[0/x]$).

We know that

$$b \in G[inl(\bullet)/x]$$

Also

$$y : X \text{ and } f : X \rightarrow G[y/x]$$

so

$$x = inr(pair(\bullet, u))$$

which means

$$x = succ(u).$$

So in fact we are doing the substitution $G[succ(u)/x]$ and, thus $g \in G[succ(u)/x]$.

We leave to the reader to carry out the simple type-checking (applying the rules for *decide* and *spread* to this definition). The interesting point is that in $\hat{g} \in G$, we are building up x either as $inl(\bullet)$ or as $inr(pair(\bullet, u))$; in one case this is $G[0/x]$ and in the other $G[succ(x)/x]$.

We have now shown that we have built the natural induction form for integers from the recursion combinator.

2.7 Co-inductive Types

The presentation of inductive types is from Nax Mendler's 1988 thesis (*Inductive Definitions in Type Theory*) and writings. Nax discovered a beautiful symmetry in the type definition process and used it to define a class of types that are sometimes called "lazy types' (see Thompson book, *Type Theory and Functional Programming* [43]). Essentially co-inductive types arise by taking maximal fixed points of monotone operators. The standard example is a *stream*, say of numbers

$$\textit{define type} \qquad S = \mathbb{N} \times S.$$

The elements of this type are objects that *generate* unbounded lists of numbers according to a certain *generation law*. The generators are recursively defined and have the form

$$\boxed{\nu\text{--}ind(a; f, z.g)}$$

much like the recursors.

Here is how the types are defined.

Definition Given a monotone operator F on types, $\nu X.F(X)$ is a type, called the *co-inductive type generated by F*.

An element of $\nu X.F(X)$ is defined from the *generator* $\nu\text{--}ind(a; f, z.b)$. This is well defined under these conditions:

Given a type D, the seed type, if $d \epsilon D$ and Y is a type such that $\nu X.F(X) \subseteq Y$ with $f : D \to Y$ and $z \epsilon D$ then $b \epsilon F(Y)$,

then $\nu - ind(d; f, z.b) \; \epsilon \; \nu X.F(X)$.

For example, if we call $\nu Y.\mathbb{N} \times Y$ a Stream, then $\nu\text{-}ind(k; f, z.\langle z, f(z+1)\rangle)$ will generate an unbounded sequence of numbers starting at k.

In order to use the elements of a co-inductive type, we need some way to force the generator to produce an element. This is done with a form called *out*. It has this property:

$$\boxed{\begin{array}{l} \text{If} \quad t \in \nu X.F(X) \\ \text{then } out(t) \in F(\nu X.F(X)). \end{array}}$$

This form obeys the following ***computation rule.***

$$\boxed{\begin{array}{l} \text{out}(\nu\text{–}ind(d; f, z.b)) \\ \text{evaluates to} \quad b[d/z, (y \mapsto \nu\text{–}ind(y; f, z.b))/f]. \end{array}}$$

2.8 Subset Types and Logic

One of the most basic and characteristic types of Nuprl is the so called *set type* or *subset type*, written $\{x : A \mid P(x)\}$ and denoting the subtype of A consisting of exactly those elements satisfying condition P. This concept is closely related to the set theory notion written the same way and denoting the "subset" of A satisfying the predicate P. In axiomatic set theory the existence of this set is guaranteed by the ***separation axiom.*** The idea is that the predicate P separates a subset of A as in the example of say the prime numbers, $\{x : \mathbb{N} \mid prime(x)\}$.

To understand this type, we need to know something about predicates. In axiomatic set theory the predicates allowed are quite restrictive; they are built from the atomic membership predicate, $x \in y$ using the first order predicate calculus over the universe of sets. In type theory we allow a different class of predicates — those involving predicative higher-order logic in a sense. This topic is discussed in many articles and books on type theory [33, 13, 36, 43, 12, 11] and is beyond the scope of this article, so here we will just assume that the reader is familiar with one account of propositions-as-types or representing logic in type theory.

The Nuprl style is to use the type of propositions, denoted *Prop*. This concept is stratified into $Prop_i$ as in *Principia Mathematica*, and it is related by the propositions-as-types principle to the large types such as *Type*. $Prop_i$ are indeed considered to be a "large types." (See [11] for an extensive discussion of this notion.) For the work we do here we only need the notions of *Type* and *Prop* which we take to be $Type_1$ and $Prop_1$ in the full Nuprl theory.