Lecture 12: Recursive Types

Chris Martens • February 20, 2026

1 Lecture outline

- Recursive definitions
- Finite vs. infinite types
- Examples: numbers, lists, and trees
- The μ type connective

2 Recursive definitions

To understand recursion, let's look it up in my favorite dictionary:
`https://acronymy.net/define/recursion`

Recursion is about self-reference. Take, for example, a recursive function:

```
fact n =
    if n = 0
    then 1
    else n * fact(n-1)
```

In the body (definition) of the function, we call (reference) the very function we're defining.
Similarly, the definition of a list is recursive:

- Informally, *a list is empty or a tuple containing data and a list*
- Formally, `list = nil | cons (data * list)`

Recursion implies a way to iteratively *unspool* the definition.
Unspooling factorial:

```
fact 3 -----> if 3 = 0 then 1 else 3 * fact 2
       -----> 3 * fact 2
       -----> 3 * (if 2 = 0 then 1 else 2 * fact 1)
       -----> 3 * (2 * fact 1)
       -----> 3 * (2 * (if 1 = 0 then 1 else 1 * fact 0))
       -----> 3 * (2 * 1 * (fact 0))
       -----> 3 * (2 * 1 * (if 0 = 0 then 1 else 0 * fact (0-1)))
       -----> 3 * 2 * 1 * 1
```

Unspooling "list":

- `(5, (4, nil)` is a list because 5 is data and...
- `(4, nil)` is a list, because 4 is data and...
- `nil` is a list.

A recursive function defines a way of unspooling computation (dynamic semantics). A recursive *type*, or data definition, defines a way of unspooling what it means for something to be well-typed. It allows us to capture *infinite* datatypes, as we will see next.

3 Infinite Types

We've seen how to use sum and product types to represent Booleans and options, building up a rich language of data with a small number of type operators. However, we've been restricted to *finite* types (in the sense of thinking of types as classes of values). Next we will think about introducing infinite types to our system.

3.1 Natural numbers

Natural numbers are a good (perhaps even canonical) place to start for infinite types. Let's think about what it means to add them to our language as a "native" type, following a by-now familiar recipe:

- Identify the *class of values* represented by the type.
- Come up with *introduction rules* for the type that generate those values
- Come up with *elimination rules* for that type that allow us to use the information it represents.

The class of values represented by Nat is the mathematical set of natural numbers $\mathbb{N}$, i.e. the smallest set that contains a distinguished element (zero) and a successor of each element in the set. We have actually already seen inference rules that do this, before we thought of them as defining a type.

$$\frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \mathsf{Nat}I/0 \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash s(e) : \mathsf{Nat}}\ \mathsf{Nat}I/s$$

What should the elimination rules be for this type? We need to pick an answer that is sound and complete for *all* ways of using the introduction rules, so for example if the eliminator has type t and is used after the intro rule for 0, it has to be reducible to a smaller term of type t; likewise if it is used after the intro rule for successor.

```
elim(0) : t ---> ?? : t
elim(s(n)) : t ---> ?? : t
```

In programming, how do we normally write functions over nats? We might have builtin ops like addition and multiplication, but how are *those* implemented?

Gödel's T language investigated exactly this question and came up with the idea of "primitive recursion", or writing functions on nats with the following schema:

- Provide a case for 0.
- Provide a case for s(n), which is permitted to refer to n as well as the result of the function on n.

This is a kind of recursive expression, which we won't define yet in full, but we will design a custom *recursor*:

$$\frac{\Gamma \vdash n : \mathsf{Nat} \quad \Gamma \vdash e_0 : \sigma \quad \Gamma, x : \mathsf{Nat}, r : \sigma \vdash e_s : \sigma}{\Gamma \vdash \mathsf{nrec}(n, e_0, x.\ e_s) : \sigma}\ \mathsf{Nat}E$$

3.2 Lists

Here are introduction rules for a list type:

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{List}\ \tau}\ \mathsf{List}I/\mathsf{nil} \qquad \frac{e_1 : \tau \quad e_2 : \mathsf{List}\ \tau}{\Gamma \vdash \mathsf{cons}(e_1, e_2) : \mathsf{List}\ \tau}\ \mathsf{List}I/\mathsf{cons}$$

What should the elimination forms be? How can we use a list?

One idea would be to project out the head and tail, but we only know we can do that for non-empty lists.

We generally need to case-analyze it and give one branch for the empty list, and another branch for the cons case.

$$\frac{\Gamma \vdash l : \mathsf{List}\ \tau \quad \Gamma \vdash e_{\mathsf{nil}} : \sigma \quad \Gamma, x : \tau, xs : \mathsf{List}\ \tau, r : \sigma \vdash e_{\mathsf{cons}} : \sigma}{\Gamma \vdash \mathsf{lrec}(l, \mathsf{nil} \Rightarrow e_{\mathsf{nil}}, \mathsf{cons}(x, xs) \Rightarrow\ e_{\mathsf{cons}}) : \sigma}\ \mathsf{List}E$$

4 Recursive Types

It seems like these types follow a general pattern, and we'd rather not have to define a custom recursor for each one. After all, in modern languages (including Plait), we can define things like lists and trees and other things with the full language of types, and still write functions that process them. So is there some way to encode them with more general type machinery?

If we try to think of representing natural numbers as a type, we want something with "infinitely many" introduction rules, one for every number. We could encode this as a kind of infinite sum type:

$$\mathsf{Nat} \triangleq \mathsf{Unit} + \mathsf{Unit} + \mathsf{Unit} + \ldots$$

where

$$\begin{aligned} 0 &\triangleq \mathsf{tag}_1\ () \\ 1 &\triangleq \mathsf{tag}_2\ (\mathsf{tag}_1\ ()) \\ 2 &\triangleq \mathsf{tag}_2\ (\mathsf{tag}_2\ (\mathsf{tag}_1\ ())) \\ &\vdots \end{aligned}$$

If we allow ourselves recursive definitions at the meta-level, then we could write

$$\begin{aligned} \mathsf{encode}(0) &= \mathsf{tag}_1\ () \\ \mathsf{encode}(\mathsf{s}(\mathsf{n})) &= \mathsf{tag}_2\ (\mathsf{encode}(n)) \end{aligned}$$

The idea behind *recursive types* is to build this kind of definition in as a type former.

4.1 The μ type former

The *class of values* C we are interested in are values of any finite type τ that include C in the set of valid types. We *build* one by wrapping an expression whose type is allowed to refer to the type we are building, and we *use* one by unwrapping it, replacing that self-reference with the closed type.

$$\frac{\Gamma \vdash e : [\mu t.\tau/t]\tau}{\Gamma \vdash \mathsf{fold}(e) : \mu t.\tau}\ \mu I \qquad \frac{\Gamma \vdash e : \mu t.\tau}{\Gamma \vdash \mathsf{unfold}(e) : [\mu t.\tau/t]\tau}\ \mu E$$

Examples:

1. Nat: $\mu t.\mathsf{Unit} + t$
2. List τ: $\mu l.\ \mathsf{Unit} + (\tau \times l)$
3. Tree τ with data at leaves? Data at nodes?