15-312 Lecture on Recursive Types

Recursive Types

> *Give a man a fish and he will eat one day; teach a man to fish and he will eat every day.*

Recursive types can be understood in the light of this proverb. When studying Gödel's T, we introduced all the tools we need to work with natural numbers: the basic constructors (z and s) and the universal destructor (natrec). In an homework, we saw that lists of natural numbers and binary trees could be defined in a similar way. We could use the same mold to define strings, lists of lists, etc. This becomes rather tedious after a while, though: we keep on defining operators that are mostly identical over and over. Even worse, each of these types is hardwired in the language: if we have designed and implemented a language with lists, adding trees amounts to designing and implementing a new language.

Recursive types capture what all these special-purpose types have in common, and provides a single construction to define them all: once we design a language with recursive types, we automatically have a language where we can define natural numbers, lists, trees, etc., whatever type of this sort we can think about.

A recursive type is a type whose definition refers to itself. With the exception of pure enumeration types (e.g., `bool`), ML datatypes are recursive[1]. Because the definition of a recursive type refers to itself, there are infinitely many terms of this type (again, think about natural numbers, lists, trees, etc.).

Syntax

$$
\begin{array}{llcl}
\textit{Types} & \tau & ::= & t \mid \mathsf{arrow}(\tau_1, \tau_2) \mid \mathsf{rec}(t.\tau) \\
\textit{Expressions} & e & ::= & x \mid \mathsf{lam}[\tau](x.e) \mid \mathsf{app}(e_1, e_2) \\
 & & \mid & \mathsf{fold}[t.\tau](e) \mid \mathsf{unfold}(e)
\end{array}
$$

We include functions in this definition because they provide a starting point for expressions (unit would perform this task equally well), and also because several interesting behaviors involve functions. They are not an essential component of recursive types, although they make using them interesting.

[1] ML datatypes also make use of other primitives, namely polymorphism (e.g., `'a list`) and type definitions (the possibility of giving a name to a type, in the same way as `let` allows us to give a name to an expression).

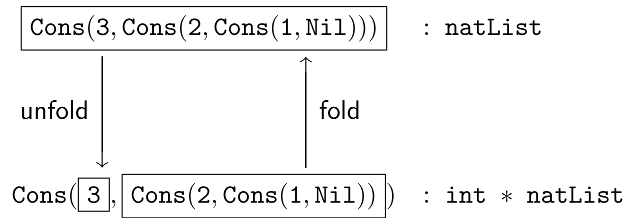
Recursive types introduce a notion of *type variable*, written t above. The recursive type $\mathsf{rec}(t.\tau)$ binds t within the type τ, which could now refer to t.

Intuition

Consider the following ML definition for lists of natural numbers:

```
datatype natList = Nil | Cons of int * natList
```

Intuitively, a list of integers such as `Cons(3,Cons(2,Cons(1,Nil)))` has two types: if we consider it as a whole, it is an expression of type `natList`; if look more closely, it is a pair consisting of the integer `3` (the head of the list) and the list of integers `Cons(2,Cons(1,Nil))` (its tail). The job of fold and unfold is to mediate these two types:



Every time we build a list from a head and a tail, we are implicitly doing a fold operation to convert the two parts into a list. Similarly, every time we look at what is inside a list, by means of pattern matching or a case construct, we are unfolding it into its definition.

We will now see what this means formally.

Typing Semantics

Because recursive types rely on binders, it is worth using a judgment to formalize when they are well formed. This judgment is $\Delta \vdash t$ type, with Δ a set of hypotheses of the form t_1 type, $\ldots, t_n$ type. It is defined as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}}\ \textbf{tp_id}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathsf{arrow}(\tau_1, \tau_2)\ \mathsf{type}}\ \textbf{tp_arrow} \qquad \frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathsf{rec}(t.\tau)\ \mathsf{type}}\ \textbf{tp_rec}$$

The typing judgment for expressions assumes the form $\Delta; \Gamma \vdash e : \tau$, where Δ is a typing context for type variables and Γ is a typing context for expression names. The typing rules for expressions is as follows. The rules for functions are as usual, with the addition of the typing context Δ which plays no role in them.

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta; \Gamma \vdash [\mathsf{rec}(t.\tau)/t]e : [\mathsf{rec}(t.\tau)/t]\tau}{\Delta; \Gamma \vdash \mathsf{fold}[t.\tau](e) : \mathsf{rec}(t.\tau)}\ \textbf{tp_fold}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{rec}(t.\tau)}{\Delta; \Gamma \vdash \mathsf{unfold}(e) : [\mathsf{rec}(t.\tau)/t]\tau}\ \textbf{tp_unfold}$$

Note that a bottom up typechecker recovers the type of e and can use it to compute the type of $\mathsf{unfold}(e)$. This is the reason why unfold does not need to be annotated with a type.

Transition Semantics

fold is the constructor of expressions of a recursive type and therefore form the basis for values of this type. unfold is the destructor and the two annihilate each other when they meet.

$$\frac{v \ \mathsf{val}}{\mathsf{fold}[t.\tau](v)\ val}\ \textbf{val_fold}$$

$$\frac{e \mapsto e'}{\mathsf{fold}[t.\tau](e) \mapsto \mathsf{fold}[t.\tau](e')}\ \textbf{step_fold} \qquad \frac{e \mapsto e'}{\mathsf{unfold}(e) \mapsto \mathsf{unfold}(e')}\ \textbf{step_unfold}$$

$$\frac{v \ \mathsf{val}}{\mathsf{unfold}(\mathsf{fold}[t.\tau](v)) \mapsto v}\ \textbf{step_unfold_fold}$$

Type Safety Theorem

The type safety results are as usual and they are proved using the techniques we have seen.

1. **Type Preservation**: If $\mathcal{T} :: \Gamma \vdash e : \tau$ and $\mathcal{E} :: e \mapsto e'$, then $\mathcal{T}' :: \Gamma \vdash e' : \tau$.
2. **Progress**: If $\mathcal{T} :: \cdot \vdash e : \tau$, then either $\mathcal{V} :: e \ \mathsf{val}$ or $\mathcal{S} :: e \mapsto e'$.

Deconstructing ML Datatypes

Consider again the above ML definition for lists of natural numbers:

```
datatype natList = Nil | Cons of int * natList
```

This definition provides us with a type (`natList`), two constructors (`Nil` and `Cons`), and a destructor (in ML it is the case statement

case e **of** `Nil =>` e_{Nil} `| Cons(n,l) =>` e_{Cons}

but this is really just the iterator $\mathsf{listrec}(e, e_{Nil}, n.l.e_{Cons})$).

Let's see what is going on under the hood now that we know about recursive types. First of all, let's model the type `natList` itself. We are going to have a variant record containing one of two things: something that we will understand as `Nil` and something else that will stand for `Cons`:

- `Nil` by itself is just a marker, an entity that carries no other information besides the fact that it is there. This suggests using unit, the nullary product type, to represent it.

- `Cons` is a pair consisting of a number and a recursive element of type `natList`.

We will use the names nil and cons as labels in a labeled variant (to distinguish them from the constructors, we use all-lowercase words). Then, in the concrete syntax, `natList` is defined as

$$\mu\ natList.[\mathsf{nil} : \mathsf{unit}; \mathsf{cons} : \mathsf{int} \times natList]$$

If we want to use the abstract syntax, we have instead

$$\mathtt{rec}(natList.\mathtt{var}[\mathsf{nil}, \mathsf{cons}](\mathsf{unit}, \mathtt{prod}(\mathsf{unit}, natList)))$$

Note that $natList$ is just a bound name here (we could have used "t") — making it available as a name in a program requires other linguistic features, type definitions. We will abbreviate either of these two type expressions as "natList" (including the quotes). In particular, when we write "natList" in an expression, we mean the above recursive type, not the type variable.

Let's now define the operations that have to do with `natList`, namely the constructors `Nil` and `Cons` and an example of the `case` statement. To make things more readable, we will write the types appearing in an expression in gray. We start with the constructors Nil and Cons.

- How do we build an empty list? Well, there is only one object of type unit, that is (). Next we need to package it as the nil-labeled entry of a variant of type [nil : unit; cons : int × "natList"], that is

  $$[\mathsf{nil} = ()]_{[\mathsf{nil:unit;cons:int}\times\text{``natList''}]}$$

  and finally we fold it into an object of type "natList":

  $$\begin{array}{l}\mathsf{fold}[natList.[\mathsf{nil} : \mathsf{unit}; \mathsf{cons} : \mathsf{int} \times natList]] \\ \quad ([\mathsf{nil} = ()]_{[\mathsf{nil:unit;cons:int}\times natList]})\end{array}$$

  Let us rewrite this using the abstract syntax:

  $$\begin{array}{l}\mathsf{fold}[natList.\mathtt{var}[\mathsf{nil}, \mathsf{cons}](\mathsf{unit}, \mathtt{prod}(\mathsf{unit}, natList))] \\ \quad (\mathsf{inj}[\mathtt{var}[\mathsf{nil}, \mathsf{cons}](\mathsf{unit}, \mathtt{prod}(\mathsf{unit}, natList)); \mathsf{nil}](\mathsf{unit}))\end{array}$$

- Cons is defined exactly in the same way, except that we need to pass it two arguments: the natural number and the list that we want to cons together. The concrete syntax is

  $$\begin{array}{l}\lambda n : \mathsf{int}.\lambda l : \text{``natList''}.\mathsf{fold}[natList.[\mathsf{nil} : \mathsf{unit}; \mathsf{cons} : \mathsf{int} \times natList]] \\ \qquad\qquad\qquad ([\mathsf{cons} = (n, l)]_{[\mathsf{nil:unit;cons:int}\times\text{``natList''}]})\end{array}$$

  and the abstract syntax is

  $$\begin{array}{l}\mathsf{lam}[\mathsf{int}](n. \\ \mathsf{lam}[\text{``natList''}](l. \\ \mathsf{fold}[\ natList.\mathsf{var}[\mathsf{nil}, \mathsf{cons}](\mathsf{unit}, \mathsf{prod}(\mathsf{unit}, natList))] \\ \quad (\mathsf{inj}[\mathsf{var}[\mathsf{nil}, \mathsf{cons}](\mathsf{unit}, \mathsf{prod}(\mathsf{unit}, \text{``natList''})); \mathsf{nil}](\mathsf{pair}(n, l)))))\end{array}$$

- We are left with the `case` statement. We will not be able to define listrec until we study polymorphism. Instead, we will look at a sample instance.

  As an example, let's define the function head (a destructor), which returns the element at the head of a list of integers l, in ML:

  $$\texttt{\textbf{case}}\ l\ \texttt{\textbf{of}}\ \texttt{Nil => 0 | Cons(n,l) =>}\ n$$

  where we are returning 0 if the list is empty (we will take more interesting actions once we work with languages with exceptions).

  By definition, l will be given to us as an expression of type "natList". Therefore, we need to unfold it to access the inner variant and then discriminate on the tag. The overall code is as follows:

  $$\begin{array}{rlll} \lambda l\!:\text{“natList”}.\mathsf{case} & (\mathsf{unfold}\ l) & & \\ \mathsf{of} & [\mathsf{nil} = ()] & => & 0 \\ | & [\mathsf{cons} = p] & => & \mathsf{fst}(p) \end{array}$$

  The corresponding abstract syntax is

  $$\begin{array}{l} \mathsf{lam}[\text{“natList”}](l. \\ \mathsf{case}[\mathsf{int}, \mathsf{int}; \mathsf{nil}, \mathsf{cons}](\mathsf{unfold}\ l, \\ \qquad u.0, \\ \qquad p.\mathsf{fst}(p)) \end{array}$$

  Other typical functions on lists, for example the tail function, or the function that checks whether a list is empty, are defined similarly.

Recursive Functions Revisited

For a moment, let's step to an untyped functional language, the simplest of them all, containing only functions and applications:

$$\textit{Untyped expressions} \qquad e \ ::= \ x \ | \ \lambda x.\, e \ | \ e_1\, e_2$$

For convenience, we are showing only the concrete syntax. Note that there is no type decoration for λ. Assume a call-by-name semantics for this language.

Now, consider the expression Y defined as follows:

$$Y \ = \ \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

(This weird expression goes by the name of *call-by-name* Y combinator).[2] Call Y' the subexpression bound by λf so that $Y = \lambda f.\, Y'$. Now, let's partially evaluate starting

[2]There is also a slightly more complicated *call-by-value* Y combinator — see Pierce's book.

it from a function $\lambda g.\, e$, where e is any expression, possibly containing g:

$$\begin{array}{rcl} Y\ (\lambda g.\, e) & = & (\lambda f.\ \underbrace{(\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))}_{Y'})\ (\lambda g.\, e) \\ & \mapsto & \underbrace{(\lambda x.\, (\lambda g.\, e)\ (x\ x))\ (\lambda x.\, (\lambda g.\, e)\ (x\ x))}_{[\lambda g.\, e/f]Y'} \\ & \mapsto & (\lambda g.\, e)\ (\underbrace{(\lambda x.\, (\lambda g.\, e)\ (x\ x))\ (\lambda x.\, (\lambda g.\, e)\ (x\ x))}_{[\lambda g.\, e/f]Y'}) \\ & \mapsto & [\underbrace{(\lambda x.\, (\lambda g.\, e)\ (x\ x))\ (\lambda x.\, (\lambda g.\, e)\ (x\ x))}_{[\lambda g.\, e/f]Y'}/g]e \end{array}$$

We get therefore that

$$[\lambda g.\, e/f]Y' \mapsto^* [[\lambda g.\, e/f]Y'/g]e$$

If we define the function recursor[3] $\mathsf{fix}(g.e)$ to be $[\lambda g.\, e/f]Y'$, this corresponds to the familiar step in the semantics of fix: i.e., $\mathsf{fix}(g.e) \mapsto [\mathsf{fix}(g.e)/g]e$.

With Y' in hand, there is no need to have an explicit recursion operator since it behaves in the desired way. The problem is that Y, and therefore Y', is not typable in a functional language without recursive types.[4] Try it!

Recursive types change this picture dramatically. Consider the function recursor $\mathsf{fix}[\tau](g.e)$ for a specific type τ (again, we haven't yet examined polymorphism). Then, we can give a type to the the Y combinator that operates on functions of type $\tau = \tau' \to \tau'$ (which is also the type of the bound variable g). Call it Y_τ:

$$\begin{array}{rcl} Y_\tau & = & \lambda f : \tau \to \tau. \qquad (\lambda x : \mu t.t \to \tau.\, f\ ((\mathsf{unfold}\ x)\ x)) \\ & & (\mathsf{fold}[\mu t.t \to \tau](\lambda x : \mu t.t \to \tau.\, f\ ((\mathsf{unfold}\ x)\ x))) \end{array}$$

Let's check if the types work:

$$\begin{array}{l} \lambda f : \tau \to \tau. \qquad \overbrace{(\lambda x : \mu t.t \to \tau.\ \overbrace{f\ \overbrace{(\overbrace{(\mathsf{unfold}\ x)}^{(\mu t.t\to\tau)\to\tau}\ \overbrace{x}^{\mu t.t\to\tau}}^{\tau}}^{\tau}\ ))}^{(\mu t.t\to\tau)\to\tau} \\ \quad \underbrace{(\mathsf{fold}[\mu t.t \to \tau]\ \underbrace{(\lambda x : \mu t.t \to \tau.\, f\ ((\mathsf{unfold}\ x)\ x)))}_{(\mu t.t\to\tau)\to\tau \text{ (see above)}}}_{\mu t.t\to\tau} \end{array}$$

Now, by applying the right part of the top line to the second, we obtain a term of type τ. Finally, by abstracting over f, we obtain $(\tau \to \tau) \to \tau$.

The type of our makeshift recursor would then be τ.

[3] In earlier lectures, we wrote this operator as rec. We now switch to fix to avoid confusion with the recursive type constructor.

[4] In general, there is no simple type τ such that $(x\ x) : \tau$

Objects Revisited

When talking about product types, we saw that objects are a form of recursive product: a standard record was extended with one extra binder in each field which stood for the entire object. Now that we have recursive types, we do not need this special construction. We can instead assemble an object directly from product types using rec.

Consider as an example the counter object adapted from Pierce's book: this counter contains one field, val, corresponding to the value of the counter itself, and two methods, inc and dec, to increment and decrement it, respectively. The type of this counter object is given by the following recursive type:

$$\mu C. \{\mathsf{val} : \mathsf{int}, \;\; \mathsf{inc} : \mathsf{unit} \to C, \;\; \mathsf{dec} : \mathsf{unit} \to C\}$$

Let's refer to this type as "counter" (again with quotes). Given an object c of type "counter", we obtain its value by unfolding it to a record and then projecting on the label val:

$$\mathsf{prj}[\mathsf{val}](\mathsf{unfold}(c))$$

To increment the counter, we similarly project on label inc and apply the resulting function to the unit value:

$$(\mathsf{prj}[\mathsf{inc}](\mathsf{unfold}(c)))\ ()$$

The value returned by this application is a new counter.

How do we create an object? We will define a function new that when applied to some number n creates an object with initial value of n. It is easiest to start with an ML-like concrete syntax extended with labeled records:

$$\begin{array}{l} \mathbf{fun}\ \ \mathsf{new}(n\text{: }\mathsf{int}) = \\ \qquad (\mathsf{val} = n, \\ \qquad\ \ \mathsf{inc} = \lambda u\text{: }\mathsf{unit}.\, \mathsf{new}(n+1), \\ \qquad\ \ \mathsf{dec} = \lambda u\text{: }\mathsf{unit}.\, \mathsf{new}(n-1)) \end{array}$$

The corresponding concrete syntax is as follows:

$$\begin{array}{l} \mathsf{fix}[\mathsf{int} \to \text{“}\mathsf{counter}\text{”}](new. \\ \quad \mathsf{lam}[\mathsf{int}](n.\{\mathsf{val} = x, \\ \qquad\qquad\quad \mathsf{inc} = \mathsf{lam}[\mathsf{unit}](new\ (n+1)) \\ \qquad\qquad\quad \mathsf{dec} = \mathsf{lam}[\mathsf{unit}](new\ (n-1))\})) \end{array}$$

Note that all recursion is encapsulated in the use of fix.