We saw how to use polynomial types in generic programming last recitation. We can extend polynomial types to inductive and coinductive types, which allows us to express more kinds of data structures.

Next week, we’ll see we can also go further to introduce polymorphic types into our programming language, which will eventually lead us to System F.

1 Inductive Types

An inductive type \( \mu(t.\tau) \) is the least type that contains \( t.\tau \). In other words, inductive types are characterized by the structure of the constructors they are built with.

Consider the following modification of System T with products and sums, which now includes inductive types.

\[
\text{Typ} \quad \tau ::= \ldots \quad \mu(t.\tau) \quad \text{inductive type}
\]

\[
\text{Exp} \quad e ::= \ldots \quad \text{fold}\{t.\tau\}(e) \quad \text{inductive fold}
\]
\[\text{rec}\{t.\tau\}(x.e_1; e_2) \quad \text{inductive recursion}
\]

Note how \texttt{nat} is not a type here anymore. This is because we can define \texttt{nat} \( \triangleq \mu(t.1 + t) \) by using inductive types instead of having a special definition of the \texttt{nat} type in our system. And likewise, all numbers can be defined in terms of inductive fold, and arithmetic can be defined in terms of inductive recursion.

Also note that \( \mu \) takes in a polynomial type operator and returns another type. If \( \phi \) is a polynomial type operator (such as \( t.1 + t \)), then \( \mu(\phi) \) is the type inductively defined by \( \phi \). In order to actually have values with these inductive types, however, we also need the \texttt{fold} and \texttt{rec} constructs to allow us to introduce and eliminate expressions of this type.

1.1 Statics

\[
\frac{e : [\mu(t.\tau)/t]\tau}{\Gamma \vdash \text{fold}\{t.\tau\}(e) : \mu(t.\tau)}
\]
\[
\frac{\Gamma, x : [\tau'/t]\tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \mu(t.\tau)}{\Gamma \vdash \text{rec}\{t.\tau\}(x.e_1; e_2) : \tau'}
\]
1.2 Dynamics

Note how we use map to apply the recursor what was inside of the fold!

\[
\begin{align*}
\text{fold}\{t.\tau\}(e) & \quad \text{val} \\
\text{rec}\{t.\tau\}(x.e_1; e_2) & \quad \text{val} \quad \text{rec}\{t.\tau\}(x.e_1; e_2') \\
\text{rec}\{t.\tau\}(x.e_1; \text{fold}\{t.\tau\}(e_2)) & \quad \text{map}\{t.\tau\}(y.\text{rec}\{t.\tau\}(x.e_1; y))(e_2)/x(e_1)
\end{align*}
\]

1.3 nat as an Inductive Types

As our very first example, we show that it’s possible to define \(\text{nat}\) as \(\mu(t.1 + t)\), by providing a definition \(z, s, (\cdot)\) and \(\text{rec}\{\cdot, \cdot\}\{\cdot, \cdot\}\). Intuitively, the type \(\text{nat}\) can be viewed as a sum type: it’s either already zero, or you can extract a predecessor out of it. In this case we make the following definitions:

\[
\begin{align*}
z & \triangleq \text{fold}\{t.1 + t\}(1 \cdot \langle \rangle) \\
s(n) & \triangleq \text{fold}\{t.1 + t\}(\tau \cdot n) \\
\text{iter}\{\tau\}(e_0; x.e_1)(n) & \triangleq \text{rec}\{t.1 + t\}(x.\text{case } \rho \{1, \tau \cdot e_0 | \tau \cdot x \mapsto e_1\}; n)
\end{align*}
\]

Now it’s very important for you to take a piece of paper and:

- See that \(z\) is typed \(\mu(1.1 + t)\).
- See that \(s(n)\) is well typed by showing \(s(n) : \mu(1 + t)\) given \(n : \mu(1 + t)\).
- See that \(\text{iter}\) is well typed under proper premises.
- Use the dynamics to convince your self \(\text{iter}\) does implements the iterator for \(\text{nat}\).

Intuitively, \(\mu\) types internalizes information when formed. As you form larger and larger values of \(\mu\) type, you are constantly “putting information into the value you formed”. Those information you internalized are accessed altogether when a \(\mu\) type is eliminated through \(\text{rec}\).

1.4 Working with Inductive Types

It can be difficult to understand what \(\text{fold}\) and \(\text{rec}\) are actually doing here. Although \(\text{fold}\) and \(\text{rec}\) are actually operators on expressions, let’s think of them as functions for a bit. If they were functions, \(\text{fold}\) and \(\text{rec}\) would have these types:

\[
\begin{align*}
\text{fold}(t.\tau) : & \quad [\mu(t.\tau)/t] \tau \rightarrow \mu(t.\tau) \\
\text{rec}(t.\tau) : & \quad ([\rho/t] \tau \rightarrow \rho) \rightarrow \mu(t.\tau) \rightarrow \rho
\end{align*}
\]

There is a lot of substitution in this view of \(\text{fold}\) and \(\text{rec}\), so let’s go through an example showing how \(\text{fold}\) and \(\text{rec}\) are actually used. We can use inductive types to define lists as you’ve seen in SML.

\[
\text{list} \triangleq \mu(t.1 + (\text{int } \times t))
\]
What then would be the type of \( \text{fold}_{\text{list}} \)?

\[
\text{fold}\{\mu (t.1 + (\text{int} \times t))\} : [\mu (t.1 + (\text{int} \times t))/t](1 + (\text{int} \times t)) \rightarrow \mu (t.1 + (\text{int} \times t)) \\
: [\text{list}/t](1 + (\text{int} \times t)) \rightarrow \text{list} \\
: 1 + (\text{int} \times \text{list}) \rightarrow \text{list}
\]

As you can see, if we give \( \text{fold}_{\text{list}} \) the empty product or the product of an \text{int} and another \text{list}, \( \text{fold}_{\text{list}} \) will give us back a new list! If we give \( \text{fold}_{\text{list}} \) the empty product, \( \text{fold}_{\text{list}} \) will give us an empty list, and if we give \( \text{fold}_{\text{list}} \) an \text{int} and a list, we’ll get back that \text{int} cons’d onto the list. The correspondence between \( \text{fold}_{\text{list}} \) and \text{nil} and \text{cons} can also be seen in the following type isomorphism:

\[
\text{fold}_{\text{list}} : (1 + (\text{nat} \times \text{list})) \rightarrow \text{list} \cong (1 \rightarrow \text{list}) \times (\text{nat} \times \text{list} \rightarrow \text{list})
\]

Let’s look at the type of \( \text{rec}_{\text{list}} \) as well. You can think of \( \rho \) as the result type of your recursive evaluation over the inductive data structure, which in this case is a list.

\[
\text{rec}\{\mu (t.1 + (\text{int} \times t))\} : (\rho/t)(1 + (\text{int} \times t) \rightarrow \rho) \rightarrow \mu (t.1 + (\text{int} \times t)) \rightarrow \rho \\
: (1 + (\text{int} \times \rho) \rightarrow \rho) \rightarrow \mu (t.1 + (\text{int} \times t)) \rightarrow \rho \\
: (1 + (\text{int} \times \rho) \rightarrow \rho) \rightarrow \text{list} \rightarrow \rho
\]

Thus, \( \text{rec}_{\text{list}} \) first takes in a function that can compute an expression of type \( \rho \) for a list, given that we already have every recursive result for the list. This function is thus like the inductive case in our original \( \text{rec} \) for natural numbers. Using this function, \( \text{rec}_{\text{list}} \) can then compute an expression of type \( \rho \) for any list. This should in fact remind you of what the \text{map} construct does in generic programming, which is actually what we will use to define the dynamics for \( \text{rec} \).

**Exercise 1**  Consider the following inductive type that represents a tree (suppose we have \text{nat} from System T):

\[
\text{tree} \triangleq \mu (t.1 + \text{nat} \times t \times t)
\]

Define a function \( \text{twotree} : \text{nat} \rightarrow \text{tree} \) that returns a full binary tree where each node is annotated with it’s height from the leaves.

**Exercise 2** Define a function \( \text{sum} : \text{tree} \rightarrow \text{nat} \) that compute the sum of all numbers in the tree.

## 2 Coinductive Types

In contrast to inductive types, a coinductive type \( \nu (t.\tau) \) is characterized by the behavior of the destructors we use to peer inside the expression.

Consider the following updated syntax, which now includes coinductive types.
2.1 Statics

\[ \text{Typ } \tau ::= \ldots \nu(t.\tau) \text{ coinductive type} \]

\[ \text{Exp } e ::= \ldots \text{gen}\{t.\tau\}(x.e_1;e_2) \text{ coinductive generation} \]

\[ \text{unfold}\{t.\tau\}(e) \text{ coinductive unfold} \]

Just like \( \mu \), \( \nu \) takes in a polynomial type operator and makes a new type. Again, although \( \text{gen} \) and \( \text{unfold} \) are not actually functions, let’s look at the types they would have if we were to define them using function types.

\[ \text{gen}(t.\tau) : (\rho \rightarrow [\rho/t]\tau) \rightarrow \rho \rightarrow \nu(t.\tau) \]

\[ \text{unfold}(t.\tau) : \nu(t.\tau) \rightarrow [\nu(t.\tau)/t]\tau \]

2.1 Statics

\[ \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x : \tau_2 \vdash e_1 : [\tau_2/t]\tau \]

\[ \Gamma \vdash \text{gen}\{t.\tau\}(x.e_1;e_2) : \nu(t.\tau) \quad \Gamma \vdash \text{unfold}\{t.\tau\}(e) : [\nu(t.\tau)/t]\tau \]

2.2 Dynamics

\[ \text{gen}\{t.\tau\}(x.e_1;e_2) \text{ val} \]

\[ e \mapsto e' \quad \text{unfold}\{t.\tau\}(e) \mapsto \text{unfold}\{t.\tau\}(e') \]

\[ \text{unfold}\{t.\tau\}\{\text{gen}\{t.\tau\}(x.e_1;e_2)\} \mapsto \text{map}\{t.\tau\}(y.\text{gen}\{t.\tau\}(x.e_1;y);[e_2/x]e_1) \]

Notice that the rule for stepping \( \text{unfold} \) is exactly the dual of the rule for stepping \( \text{rec} \).

This is pretty abstract, so let’s look at using \( \text{gen} \) and \( \text{unfold} \) with the coinductive interpretation of int lists. As you’ve seen in 15-150 and in lecture yesterday, streams can be used to encode an infinite data structure. Using coinductive types, we can define lists containing ints that are kind of like finite streams.

\[ \text{colist} \triangleq \nu(t.1 + (\text{nat} \times t)) \]

What then would be the type of \( \text{gen}_{\text{colist}} \)?

\[ \text{gen}\{t.1 + (\text{nat} \times t)\} : (\rho \rightarrow [\rho/t](1 + (\text{nat} \times t))) \rightarrow \rho \rightarrow \nu(t.1 + (\text{nat} \times t)) \]

\[ : (\rho \rightarrow (1 + (\text{nat} \times \rho))) \rightarrow \rho \rightarrow \nu(t.1 + (\text{nat} \times t)) \]

\[ : (\rho \rightarrow (1 + (\text{nat} \times \rho))) \rightarrow \rho \rightarrow \text{stream} \]

You can think of \( \rho \) as the type of your state, so the function given to \( \text{gen} \) of the type \( \rho \rightarrow (1 + (\text{nat} \times \rho)) \) is kind of like a state transition function or state automaton. Once given an expression of type \( \rho \) representing the current state, we can use the function to get either 1 (representing the end of the colist) or the next number in the stream (the \( \text{nat} \) in the product) and the next state (the \( \rho \) in the product). You can then think of the \( \rho \) in the middle of the type for \( \text{gen} \) as the “seed state” of the stream. The way \( \text{gen} \) relies on a function to get the next “state” of the stream is also analogous to how \( \text{rec} \) relies on a function to compute a final result based on previous inductive cases.
2.3 conat as a Coinductive Type

Let’s now look at the type of \( \text{unfold}_{\text{colist}} \).
\[
\text{unfold}(t.1 + (\text{nat} \times t)) : \nu(t.1 + (\text{nat} \times t)) \rightarrow [\nu(t.1 + (\text{nat} \times t))/t](1 + (\text{nat} \times t))
\]
\[
: \nu(t.1 + (\text{nat} \times t)) \rightarrow (1 + (\text{nat} \times \nu(t.1 + (\text{nat} \times t))))
\]
\[
: \text{colist} \rightarrow (1 + (\text{nat} \times \text{colist}))
\]

Note that we can have two cases for the result of \( \text{unfold}_{\text{colist}} \). In the first case, we get back the empty product, which represents the end of the list. In the second case, we get a product where the left projection corresponds to the head of the list and the right projection corresponds to the tail of the list. In this way, \( \text{unfold}_{\text{colist}} \) is able to express both of the operations we need to interact with the coinductive interpretation of lists.

2.3 conat as a Coinductive Type

It’s worthwhile to note the duality between \( \mu \) and \( \nu \) types: for an \( \mu \)-type:

- Values of inductive types are formed from a “limited form”. I.e., the type operator specifies what is internalized into a \( \mu \) type by allowing only values of a specific form to be turned into a value of \( \mu \) type.
- But you are free to choose what to do with a value of \( \mu \) type. The elimination form doesn’t care what comes out an elimination.
- Values are “internalized” into \( \mu \) type during introduction.

Conversely

- Values of coinductive types are eliminated into a “limited form”. I.e., the type operator specifies what information may be extracted from a value of coinductive type.
- But you are free to choose how such information is obtained. The introduction form does not care where you derive the information from.
- Values are “generated” from \( \nu \) type during elimination.

As a concrete example, let’s take a look at the type \( \text{conat} \triangleq \nu(t.1 + t) \). A value of \( \text{conat} \) type, when eliminated using \( \text{unfold} \), tells you whether it’s zero (left case) or a successor of another \( \text{conat} \triangleq \nu(t.1 + t) \).

- A value \( z \) of \( \text{conat} \) type should behave as the following: \( \text{unfold}(t.1 + t)(z) \rightarrow^* 1 \cdot 1 \)

- Other values \( n \) of \( \text{conat} \) type should: \( \text{unfold}(t.1 + t)(n) \rightarrow^* r \cdot n' \) where \( n' : \text{conat} \).

- A function \( s \) of type \( \text{conat} \rightarrow \text{conat} \) that returns a successor of its argument.

We provide one of a number of possible definitions:
\[
z \triangleq \text{gen}(t.1 + t)(x.1 \cdot 1); ()
\]
\[
s(n) \triangleq \text{gen}(t.1 + t)(x.\text{case } x \{ 1 \cdot \rightarrow 1 \cdot () | r \cdot n' \rightarrow \text{unfold}(t.1 + t)(n') \}; r \cdot n)
\]

Now every value of \( \text{nat} \) type has to be obtained either by \( z \) or by applying successors. In this sense the \( \mu \) type defines a least type. On the contrary, not every value of \( \text{conat} \) has to be obtained from \( z \) and successor. In particular:
• There exists a number of other values that also behaves like \( z \): 
\[
\text{gen}\{t.1 + t\}(x.1 \cdot x; \langle \rangle), \\
\text{gen}\{t.1 + t\}(x.1 \cdot \langle \rangle; \lambda (x : \text{nat}) x)
\]

• There exists an “infinite” stack of successors \( \omega \triangleq \text{gen}\{t.1 + t\}(x.r \cdot x; \langle \rangle) \). Check for yourself that unfolding this terms gives back itself.

Another way to characterize \( \text{conat} \) being “larger” than \( \text{nat} \) is to show it’s possible to turn every \( \text{nat} \) into a \( \text{conat} \):

\[
i : \text{nat} \to \text{conat} \triangleq \lambda (n : \mu (t.1 + t)) \text{rec}\{t.1 + t\}(x.\text{case } x \{ l \cdot \_ \to z | r \cdot n' \to s(n') \}; n)
\]

where \( z \) and \( s(\cdot) \) are arbitrary well-behaved definitions in context of \( \text{conat} \).

### 3 System F

Inductive and coinductive types expand the expressivity of \( T \) considerably. The power of type operators allows us to genericly manipulate data of heterogeneous types, building new types from old ones. But to write truly “generic” programs, we want truly polymorphic expressions—functions that operate on containers of some arbitrary type, for example. To gain this power, we add parametric polymorphism to the language, which results in System \( F \), introduced by Girard (1972) and Reynolds (1974).

\[
\begin{align*}
\text{Typ} & \quad \tau \ ::= \quad t \quad \text{type variable} \\
& \quad \tau_1 \to \tau_2 \quad \text{function} \\
& \quad \forall (t.\tau) \quad \text{universal type} \\
\text{Exp} & \quad e \ ::= \quad x \quad \text{variable} \\
& \quad \lambda (x : \tau) e \quad \text{abstraction} \\
& \quad e_1(e_2) \quad \text{application} \\
& \quad \Lambda(t) e \quad \text{type abstraction} \\
& \quad e[\tau] \quad \text{type application}
\end{align*}
\]

Take stock of what we’ve added since last time, and what we’ve removed. The familiar type variables are now baked into the language, along with the universal type. We also have a new form of lambda expression, one that works over type variables rather than expression variables.

What’s missing? Nearly every other construct we’ve come to know and love! As will be the case repeatedly in the course, our tools such as products, sums, and inductive types are subsumed by the new polymorphic types. The result is an extremely simple System \( F \) that is actually even more powerful.

### 3.1 Statics

Now that types have variables, we need to decide which type abt’s are considered valid. We introduce the following judgment:

\[
\Delta \vdash \tau \quad \text{type}
\]

meaning that in the type context \( \Delta \), \( \tau \) is a valid type. The type context \( \Delta \) contains the type variables that we have seen so far.
We also attach the type context to the typing judgment, which now looks like:

$$\Delta, \Gamma \vdash e : \tau$$

To define what types are valid, we essentially just want to state that closed types (ones with no free variables) are valid, and open types are invalid. These rules express that fact:

$$\Delta, \tau \text{ type } \vdash \tau \text{ type } \quad \Delta \vdash \tau_1 \text{ type } \quad \Delta \vdash \tau_2 \text{ type } \quad \Delta, t \text{ type } \vdash \tau \text{ type }$$

And now we may define the typing judgment. The cases for variable, lambda, and application are as they were in System T; we simply carry $$\Delta$$ along for the ride. There are two interesting new rules:

$$\Delta, t \text{ type, } \Gamma \vdash e : \tau \quad \Delta, t \text{ type, } \Gamma \vdash \Lambda(t) e : \forall t.\tau \quad \Delta, t \text{ type, } \Gamma \vdash e^{[\tau]} : [\tau/t]\tau'$$

Type lambdas are the introduction of universal types, and type applications are their elimination. The type application rule saying that if some expression $$e$$ is valid for all choices of $$t$$, then it will also be valid when the actual type $$\tau$$ is substituted for $$t$$ (provided that $$\tau$$ is a valid type).

This is very similar to polymorphic types in ML, where types may contain type variables. Be aware that ML usually leaves the type lambda implicit. That is, the ML type

$$('a \rightarrow 'b) \rightarrow 'c$$

is actually

$$\forall \alpha.\forall \beta.\forall \gamma. (\alpha \rightarrow \beta) \rightarrow \gamma$$

in System F. Observe that ML implicitly places the type lambdas at the front of the type. As we will soon see, this is an important distinction between ML and System F. ML cannot directly express a type like

$$\forall \alpha.\alpha \rightarrow \forall \beta.\beta$$

which System F easily can do.

ML also does not explicitly apply types. Consider the polymorphic identity function in F:

$$\text{id} \triangleq \forall \alpha.\lambda(x : \alpha) x$$

This function is truly polymorphic, as we can apply $$\text{id}[\text{nat}]$$ to get the identity function on naturals, $$\text{id}[\text{nat} \rightarrow \text{nat}]$$ to get the identity function on functions from naturals to naturals, etc. However, in ML, the type checker automatically applies the appropriate type argument to its type abstractions. $$\text{id} \ 0$$ and $$\text{id} \ (\text{fn} \ (x:\text{nat}) \Rightarrow x)$$ implicitly involve the specialization of the function id.
3.2 Dynamics

System F also has a remarkably simple dynamics. The rules for lambda and application remain the same as in lazy/eager System T, and we need only introduce the rules for type lambda and type application.

\[
\begin{align*}
\Lambda(t) e \text{ val} & \quad e \mapsto e' \\
\Lambda(t) e[\tau] & \mapsto e'[\tau] \\
\Lambda(t) e[\tau] & \mapsto [\tau/t] e
\end{align*}
\]

That’s it! Type functions are values, type applications are eager, and they eventually substitute a type for a variable in a type abstraction.

Examples:

\[
\Lambda(\alpha) \lambda (x : \alpha) x \text{ is the polymorphic identity function}
\]

\[
\Lambda(\alpha) \Lambda(\beta) \lambda (f : \alpha \to \beta) \lambda (x : \alpha) f(x) \text{ is the polymorphic applicator function}
\]

3.3 Definability in System F

System F, though deceivingly simple at first glance, is powerful enough express a number of ideas we have previously discussed. In particular, they includes unit, void, sums, products and sums. Here we demonstrate how this is done:

**unit Type** The unit type have one exactly value. There is no elimination rule for unit because a value of unit type intuitively internalized no information content. This motivates us to make the following definition:

\[
\text{unit} \triangleq \forall(\alpha.\alpha \to \alpha) \\
\langle \rangle \triangleq \Lambda(\alpha) \lambda(x : \alpha) x
\]

**void Type** The void type contains no values at all. It has no introduction rule but only elimination rules. The elimination can be typed arbitrarily, this motivates us to make the following definition:

\[
\text{void} \triangleq \forall(\alpha.\alpha) \\
\text{case } e \{\} : \tau \triangleq e[\tau]
\]

**+ Type** As you may have noticed in lambda calculus, encoding things is more about elimination than introduction: we carefully identify how a value of a encoded type may be used, and try to encode the type with another type that internalizes enough information for it’s elimination. This motivates us to look at the elimination rule of sum types:

\[
\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{case } e \{1 : x \leftrightarrow e_1 \mid r : y \leftrightarrow e_2\} : \tau
\]

Essentially, to eliminate a + type and reach a result of \(\tau\), two pieces of information are need: what to do with a \(\tau_1\) (a function of type \(\tau_1 \to \tau\)) and what to do with a \(\tau_2\) (a function of type \(\tau_2 \to \tau\)). A value of a + type can then be viewed as something that produce the desired result...
of type \( \tau \), provided with two functions of type \( \tau_1 \to \tau \) and \( \tau_2 \to \tau \). This motivates us to make the following definitions:

\[
\tau_1 + \tau_2 \triangleq \forall r.((\tau_1 \to r) \to (\tau_2 \to r) \to r)
\]

\[
\text{case } e \{ 1 \cdot x \mapsto e_1 | r \cdot y \mapsto e_2 \} : \tau \triangleq e[\tau](\lambda (x : \tau_1) e_1)(\lambda (y : \tau_1) e_2)
\]

In the introduction rule, we need to prepare a value that will act correctly when provided with sufficient information:

\[
1 \cdot e : \tau_1 + \tau_2 \triangleq \Lambda (r) \lambda (f_l : \tau_1 \to r) \lambda (f_r : \tau_2 \to r) f_l(e)
\]

\[
r \cdot e : \tau_1 + \tau_2 \triangleq \Lambda (r) \lambda (f_l : \tau_1 \to r) \lambda (f_r : \tau_2 \to r) f_r(e)
\]

\( \times \text{ Type } \) It’s a bit of trickier for product types because our elimination rules for product type does not look like that for sum types. However this can be easily overcome: \( \tau_1 \times \tau_2 \) internalized two pieces of information, and it may be eliminated for a result of type \( r \) given a function that takes values of both types and returns a value of type \( r \). This motivates us to make the following definitions:

\[
\tau_1 \times \tau_2 \triangleq \forall r.((\tau_1 \to \tau_2 \to r) \to r)
\]

\[
\langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \triangleq \Lambda (r) \lambda (f : \tau_1 \to \tau_2 \to r) f(e_1)(e_2)
\]

\[
e \cdot 1 : \tau_1 \triangleq e[\tau_1](\lambda (x : \tau_1) x)
\]

\[
e \cdot r : \tau_1 \triangleq e[\tau_1](\lambda (x : \tau_1) \lambda (y : \tau_2) y)
\]