Recitation 5:
Inductive and Coinductive Types

15-312: Foundations of Programming Languages

Serena Wang

February 14, 2018

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.

Typ	τ	::=	$\tau_1 \to \tau_2$	function
			1	unit
			$\tau_1 \times \tau_2$	product
			0	void
			$\tau_1 + \tau_2$	sum
			$\mu(t.\tau)$	inductive type
Exp	e	::=	x	variable
			$\lambda\,(x : \tau)\,e$	abstraction
			$e_1(e_2)$	application
			$()$	empty pair
			(e_1, e_2)	pair
			$e \cdot \mathtt{l}$	left projection
			$e \cdot \mathtt{r}$	right projection
			$\mathtt{l} \cdot e$	left injection
			$\mathtt{r} \cdot e$	right injection
			$\mathtt{case}\ e\ \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\}$	case
			$\mathtt{fold}\{t.\tau\}(e)$	inductive fold
			$\mathtt{rec}\{t.\tau\}(x.e_1; e_2)$	inductive recursion

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

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

$$\begin{aligned}\texttt{fold}(t.\tau) &: [\mu(t.\tau)/t]\tau \to \mu(t.\tau) \\ \texttt{rec}(t.\tau) &: ([\rho/t]\tau \to \rho) \to \mu(t.\tau) \to \rho\end{aligned}$$

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

$$\texttt{list} \triangleq \mu(t.1 + (\texttt{int} \times t))$$

What then would be the type of $\texttt{fold}_{\texttt{list}}$?

$$\begin{aligned}\texttt{fold}\{\mu(t.1 + (\texttt{int} \times t))\} &: [\mu(t.1 + (\texttt{int} \times t))/t](1 + (\texttt{int} \times t)) \to \mu(t.1 + (\texttt{int} \times t)) \\ &: [\texttt{list}/t](1 + (\texttt{int} \times \texttt{list})) \to \texttt{list} \\ &: 1 + (\texttt{int} \times \texttt{list}) \to \texttt{list}\end{aligned}$$

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

$$\texttt{fold}_{\texttt{list}} : (1 + (\texttt{nat} \times \texttt{list})) \to \texttt{list} \cong (1 \to \texttt{list}) \times (\texttt{nat} \times \texttt{list} \to \texttt{list})$$

Let's look at the type of $\texttt{rec}_{\texttt{list}}$ as well. You can think of ρ as the result type of your recursive evaluation over the inductive data structure, which in this case is a list.

$$\begin{aligned}\texttt{rec}\{\mu(t.1 + (\texttt{int} \times t))\} &: ([\rho/t]1 + (\texttt{int} \times t) \to \rho) \to \mu(t.1 + (\texttt{int} \times t)) \to \rho \\ &: (1 + (\texttt{int} \times \rho) \to \rho) \to \mu(t.1 + (\texttt{int} \times t)) \to \rho \\ &: (1 + (\texttt{int} \times \rho) \to \rho) \to \texttt{list} \to \rho\end{aligned}$$

Thus, $\texttt{rec}_{\texttt{list}}$ first takes in a function that can compute an expression of type ρ 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 `rec` for natural numbers. Using this function, $\texttt{rec}_{\texttt{list}}$ can then compute an expression of type ρ for any list. This should in fact remind you of what the `map` construct does in generic programming, which is actually what we will use to define the dynamics for `rec`.

1.1 Statics

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

1.2 Dynamics

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

$$\frac{}{\texttt{fold}\{t.\tau\}(e)\ \mathsf{val}}$$

$$\frac{e_2 \longmapsto e_2'}{\texttt{rec}\{t.\tau\}(x.e_1; e_2) \longmapsto \texttt{rec}\{t.\tau\}(x.e_1; e_2')}$$

$$\frac{}{\texttt{rec}\{t.\tau\}(x.e_1; \texttt{fold}\{t.\tau\}(e_2)) \longmapsto [\texttt{map}\{t.\tau\}(y.\texttt{rec}\{t.\tau\}(x.e_1; y))(e_2)/x](e_1)}$$

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.

Typ	τ	::=	$\ldots$	
			$\nu(t.\tau)$	coinductive type
Exp	e	::=	$\ldots$	
			$\texttt{gen}\{t.\tau\}(x.e_1; e_2)$	coinductive generation
			$\texttt{unfold}\{t.\tau\}(e)$	coinductive unfold

Just like μ, ν takes in a polynomial type operator and makes a new type. Again, although `gen` and `unfold` are not actually functions, let's look at the types they would have if we were to define them using function types.

$$\texttt{gen}(t.\tau) : (\rho \to [\rho/t]\tau) \to \rho \to \nu(t.\tau)$$
$$\texttt{unfold}(t.\tau) : \nu(t.\tau) \to [\nu(t.\tau)/t]\tau$$

This is pretty abstract, so let's look at using `gen` and `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 `int`s that are kind of like finite streams.

$$\texttt{colist} \triangleq \nu(t.1 + (\texttt{nat} \times t))$$

What then would be the type of $\texttt{gen}_{\texttt{colist}}$?

$$\begin{aligned}\texttt{gen}\{t.1 + (\texttt{nat} \times t)\} &: (\rho \to [\rho/t](1 + (\texttt{nat} \times t))) \to \rho \to \nu(t.1 + (\texttt{nat} \times t)) \\ &: (\rho \to (1 + (\texttt{nat} \times \rho))) \to \rho \to \nu(t.1 + (\texttt{nat} \times t)) \\ &: (\rho \to (1 + (\texttt{nat} \times \rho))) \to \rho \to \texttt{stream}\end{aligned}$$

You can think of ρ as the type of your state, so the function given to `gen` of the type $\rho \to (1 + (\texttt{nat} \times \rho))$ is kind of like a state transition function or state automaton. Once given an expression of type ρ 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 `nat` in the product) and the next state (the ρ in the product). You can then think of the ρ in the middle of the type for `gen` as the "seed state" of the stream. The way `gen` relies on a function to get the next "state" of the stream is also analagous to how `rec` relies on a function to compute a final result based on previous inductive cases.

Let's now look at the type of $\texttt{unfold}_{\texttt{colist}}$.

$$\begin{aligned}\texttt{unfold}(t.1 + (\texttt{nat} \times t)) &: \nu(t.1 + (\texttt{nat} \times t)) \to [\nu(t.1 + (\texttt{nat} \times t))/t](1 + (\texttt{nat} \times t)) \\ &: \nu(t.1 + (\texttt{nat} \times t)) \to (1 + (\texttt{nat} \times \nu(t.1 + (\texttt{nat} \times t)))) \\ &: \texttt{colist} \to (1 + (\texttt{nat} \times \texttt{colist}))\end{aligned}$$

Note that we can have two cases for the result of $\texttt{unfold}_{\texttt{colist}}$. 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, $\texttt{unfold}_{\texttt{colist}}$ is able to express both of the operations we need to interact with the coinductive interpretation of lists.

2.1 Statics

$$\frac{\Gamma \vdash e_2 : \tau_2 \quad \Gamma, x : \tau_2 \vdash e_1 : [\tau_2/t]\tau}{\Gamma \vdash \texttt{gen}\{t.\tau\}(x.e_1; e_2) : \nu(t.\tau)} \qquad \frac{\Gamma \vdash e : \nu(t.\tau)}{\Gamma \vdash \texttt{unfold}\{t.\tau\}(e) : [\nu(t.\tau)/t]\tau}$$

2.2 Dynamics

$$\frac{}{\texttt{gen}\{t.\tau\}(x.e_1; e_2)\ \mathsf{val}} \qquad \frac{e \longmapsto e'}{\texttt{unfold}\{t.\tau\}(e) \longmapsto \texttt{unfold}\{t.\tau\}(e')}$$

$$\frac{}{\texttt{unfold}\{t.\tau\}(\texttt{gen}\{t.\tau\}(x.e_1; e_2)) \longmapsto \texttt{map}\{t.\tau\}(y.\texttt{gen}\{t.\tau\}(x.e_1; y))([e_2/x]e_1)}$$

Notice that the rule for stepping `unfold` is exactly the dual of the rule for stepping `rec`.