CS520, Handout 26, 4 December 2006

Recursive Types

Reading: [Pierce, Chapter 20]

Syntax of Types

(compare with [Handout 17, page 3], new shown in red)

T ::=		*types*
	$\mathtt{Nat} \mid \mathtt{Bool}$	*base type*
	X	*type variable*
	$\mathtt{Unit}$	*unit type*
	$T \to T$	*function type*
	$T \times T$	*product type*
	$T + T$	*sum type*
	$\mu X.T$	*recursive type*

Example

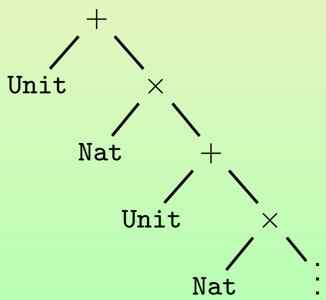
Define the type NatLst as follows:

$$\texttt{NatLst} \triangleq \mu \texttt{X}.\texttt{Unit} + (\texttt{Nat} \times \texttt{X})$$

This means NatLst is the unique type (without recursion), necessarily infinite, satisfying the equation:

$$\texttt{NatLst} = \texttt{Unit} + (\texttt{Nat} \times \texttt{NatLst})$$

The parsee tree of such an infinite type is:



Example (continued)

We now want to define the following operations:

$$\begin{aligned}
&\texttt{nil} &&: \texttt{NatLst} \\
&\texttt{cons} &&: \texttt{Nat} \longrightarrow \texttt{NatLst} \longrightarrow \texttt{NatLst} \\
&\texttt{head} &&: \texttt{NatLst} \longrightarrow \texttt{Nat} \\
&\texttt{tail} &&: \texttt{NatLst} \longrightarrow \texttt{NatLst} \\
&\texttt{isnil} &&: \texttt{NatLst} \longrightarrow \texttt{Bool}
\end{aligned}$$

This will allow us to take lists (over the natural numbers) as syntactic sugar in the presence of Unit, Nat, sum types, product types and recursive types.

Example (continued) We now want to define the following operations:

$$
\begin{aligned}
&\texttt{nil} &&: \texttt{NatLst} \\
&\texttt{cons} &&: \texttt{Nat} \longrightarrow \texttt{NatLst} \longrightarrow \texttt{NatLst} \\
&\texttt{head} &&: \texttt{NatLst} \longrightarrow \texttt{Nat} \\
&\texttt{tail} &&: \texttt{NatLst} \longrightarrow \texttt{NatLst} \\
&\texttt{isnil} &&: \texttt{NatLst} \longrightarrow \texttt{Bool}
\end{aligned}
$$

This will allow us to take lists (over the natural numbers) as syntactic sugar in the presence of `Unit`, `Nat`, sum types, product types and recursive types.

$$
\begin{aligned}
&\texttt{nil} &&= \texttt{inl}^{\texttt{Unit},\texttt{Nat}\times\texttt{NatLst}}\ \texttt{unit} \\
&\texttt{cons} &&= \lambda\texttt{n} : \texttt{Nat}.\ \lambda\ell : \texttt{NatLst}.\ \texttt{inr}^{\texttt{Unit},\texttt{Nat}\times\texttt{NatLst}}\ \{\texttt{n}, \ell\} \\
&\texttt{head} &&= \lambda\ell : \texttt{NatLst}.\ \texttt{case}^{\texttt{Unit},\texttt{Nat}\times\texttt{NatLst},\texttt{Nat}}\ \ell\ (\lambda\texttt{x} : \texttt{Unit}.\ 0)\ (\lambda\texttt{y} : \texttt{Nat}\times\texttt{NatLst}.\ (\texttt{y}.1)) \\
&\texttt{tail} &&= \lambda\ell : \texttt{NatLst}.\ \texttt{case}^{\texttt{Unit},\texttt{Nat}\times\texttt{NatLst},\texttt{NatLst}}\ \ell\ (\lambda\texttt{x} : \texttt{Unit}.\ \ell)\ (\lambda\texttt{y} : \texttt{Nat}\times\texttt{NatLst}.\ (\texttt{y}.2)) \\
&\texttt{isnil} &&= \lambda\ell : \texttt{NatLst}.\ \texttt{case}^{\texttt{Unit},\texttt{Nat}\times\texttt{NatLst},\texttt{Bool}}\ \ell\ (\lambda\texttt{x} : \texttt{Unit}.\ \texttt{true})\ (\lambda\texttt{y} : \texttt{Nat}\times\texttt{NatLst}.\ \texttt{false})
\end{aligned}
$$

Exercise: Type-check the 5 preceding function definitions.

Example (continued)

The preceding definitions can be used to add up all the elements in a list of natural numbers. This operation can be defined recursively as follows:

$$\texttt{sumlist} = \lambda\ell : \texttt{NatLst}.\ \texttt{if}\ \ (\texttt{isnil}\,\ell)\ \ \texttt{then}\ \ 0\ \ \texttt{else}\ \ (\texttt{plus}\,(\texttt{head}\,\ell)\,(\texttt{sumlist}\,(\texttt{tail}\,\ell)))$$

$$> \texttt{sumlist} :\ \ \texttt{NatLst} \longrightarrow \texttt{Nat}$$

Equivalently, we can use the fixed-point constructor, to define the same function non-recursively:

$$
\begin{aligned}
\texttt{sumlist} = \texttt{fix}\, \Big(&\lambda\texttt{F} : \texttt{NatLst} \longrightarrow \texttt{Nat}.\ \lambda\ell : \texttt{NatLst}. \\
&\qquad \texttt{if}\ \ (\texttt{isnil}\,\ell)\ \ \texttt{then}\ \ 0\ \ \texttt{else}\ \ (\texttt{plus}\,(\texttt{head}\,\ell)\,(\texttt{F}\,(\texttt{tail}\,\ell)))\Big)
\end{aligned}
$$

$$> \texttt{sumlist} :\ \ \texttt{NatLst} \longrightarrow \texttt{Nat}$$

Example (continued)

The preceding definitions can be used to add up all the elements in a list of natural numbers. This operation can be defined recursively as follows:

$$\texttt{sumlist} = \lambda\ell : \texttt{NatLst}.\ \texttt{if}\ (\texttt{isnil}\,\ell)\ \texttt{then}\ 0\ \texttt{else}\ (\texttt{plus}\,(\texttt{head}\,\ell)\,(\texttt{sumlist}\,(\texttt{tail}\,\ell)))$$

$> \texttt{sumlist} :\ \texttt{NatLst} \to \texttt{Nat}$

Equivalently, we can use the fixed-point constructor, to define the same function non-recursively:

$$\texttt{sumlist} = \texttt{fix}\Big(\lambda\texttt{F} : \texttt{NatLst} \to \texttt{Nat}.\, \lambda\ell : \texttt{NatLst}.$$
$$\texttt{if}\ (\texttt{isnil}\,\ell)\ \texttt{then}\ 0\ \texttt{else}\ (\texttt{plus}\,(\texttt{head}\,\ell)\,(\texttt{F}\,(\texttt{tail}\,\ell)))\Big)$$

$> \texttt{sumlist} :\ \texttt{NatLst} \to \texttt{Nat}$

Remark: In the program above, the function "`plus`" is not a prim op in the language. But it can be programmed using the language of typed arithmetic expressions [Pierce, Ch 8] + simply-typed lambda-calculus [Pierce, Ch 9] + "`fix`". See [Pierce, Exercise 11.11.1].

Two different views of recursive types. For the differences, see [Pierce, page 276]:

1. The **equi-recursive** approach.
2. The **iso-recursive** approach.

Pierce's presentation in Chapter 20 is based on the iso-recursive approach, see [Pierce, Fig 20-1]. Pierce's presentation in Chapter 21 is based on the equi-recursive approach.

Two different views of recursive types. For the differences, see [Pierce, page 276]:

1. The **equi-recursive** approach.
2. The **iso-recursive** approach.

Pierce's presentation in Chapter 20 is based on the iso-recursive approach, see [Pierce, Fig 20-1]. Pierce's presentation in Chapter 21 is based on the equi-recursive approach.

Our presentation is different from these two, although closer to the equi-recursive approach. The equi-recursive approach makes explicit use of the "infinite interpretation" of recursive types, our approach does not. One of the difficulties with the "infinite interpretation" of recursive types is that we are no longer able to use induction in proofs. For our presentation:

Syntax of Terms Same as in [Handout 17, page 4].

Typing Rules Same as in [Handout 17, page 5] except for T-APP and T-FIX.

Evaluation Rules Same as in [Handout 17, page 7].

In preparation for the adjusted rules T-APP and T-FIX, we define **two auxiliary function** on types, $\mathbf{in}(\ )$ and $\mathbf{out}(\ )$:

$$\mathbf{in}(T) = \begin{cases} T_1 & \text{if } T = T_1 \to T_2, \\ \mathbf{in}([X \mapsto T]\,T') & \text{if } T = \mu X.T', \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$$\mathbf{out}(T) = \begin{cases} T_2 & \text{if } T = T_1 \to T_2, \\ \mathbf{out}([X \mapsto T]\,T') & \text{if } T = \mu X.T', \\ \text{undefined} & \text{otherwise,} \end{cases}$$

In preparation for the adjusted rules T-APP and T-FIX, we define **two auxiliary function** on types, $\mathbf{in}(\,)$ and $\mathbf{out}(\,)$:

$$\mathbf{in}(T) = \begin{cases} T_1 & \text{if } T = T_1 \to T_2, \\ \mathbf{in}([X \mapsto T]T') & \text{if } T = \mu X.T', \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$$\mathbf{out}(T) = \begin{cases} T_2 & \text{if } T = T_1 \to T_2, \\ \mathbf{out}([X \mapsto T]T') & \text{if } T = \mu X.T', \\ \text{undefined} & \text{otherwise,} \end{cases}$$

and **one auxiliary relation** $\approx$ on types, which is the least congruence satisfying:

1. $T \approx T'$ where T' is obtained from T by renaming bound variables.
2. $\mu X.T \approx [X \mapsto \mu X.T]T$.

("Congruence relations" are defined in Handout 27.)

Adjusted rules T-APP and T-FIX:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \mathbf{in}(T_1) \approx T_2}{\Gamma \vdash t_1\ t_2 : \mathbf{out}(T_1)} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : T_1 \to T_2 \qquad T_1 \approx T_2}{\Gamma \vdash \mathtt{fix}\ t : T_2} \quad \text{(T-FIX)}$$

Example

Is the term "$\mathtt{fix}\,(\lambda\mathtt{f}.\,\lambda\mathtt{x}.\,(\mathtt{f}\;\mathtt{x})\;)$" typable? Let $T = (\mu X.\mathtt{Nat} \to X)$ so that:

$$\begin{aligned}
\mathbf{in}(T) &= \mathtt{Nat} \\
\mathbf{out}(T) &= T \\
\mathtt{Nat} \to T &\approx T
\end{aligned}$$

Example

Is the term "$\mathtt{fix}\,(\lambda\mathtt{f}.\,\lambda\mathtt{x}.\,(\mathtt{f}\;\mathtt{x})\;)$" typable? Let $T = (\mu X.\mathtt{Nat} \to X)$ so that:

$$\begin{aligned}
\mathbf{in}(T) &= \mathtt{Nat} \\
\mathbf{out}(T) &= T \\
\mathtt{Nat} \to T &\approx T
\end{aligned}$$

A typing of "$\mathtt{fix}\,(\lambda\mathtt{f}.\,\lambda\mathtt{x}.\,(\mathtt{f}\;\mathtt{x})\;)$" using recursive types:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{f} : T, \mathtt{x} : \mathtt{Nat} \vdash \mathtt{f} : T} \qquad \dfrac{}{\mathtt{f} : T, \mathtt{x} : \mathtt{Nat} \vdash \mathtt{x} : \mathtt{Nat}} \qquad \mathbf{in}(T) \approx \mathtt{Nat}}{\mathtt{f} : T, \mathtt{x} : \mathtt{Nat} \vdash \mathtt{f}\;\mathtt{x} : T}}{\mathtt{f} : T \vdash (\lambda\mathtt{x} : \mathtt{Nat}.\,\mathtt{f}\;\mathtt{x}\;) : \mathtt{Nat} \to T}}{\vdash (\lambda\mathtt{f} : T.\,\lambda\mathtt{x} : \mathtt{Nat}.\,\mathtt{f}\;\mathtt{x}\;) : T \to \mathtt{Nat} \to T} \qquad T \approx \mathtt{Nat} \to T}{\vdash \mathtt{fix}\,(\lambda\mathtt{f} : T.\,\lambda\mathtt{x} : \mathtt{Nat}.\,\mathtt{f}\;\mathtt{x}\;) : \mathtt{Nat} \to T}$$

HD 26 Recursive Types – p.9/9