Recursive Types

Lecture 17
CS 565
4/9/09
Recursive Types

- It is useful to be able to define recursive data structures.
- As the name suggests, a recursive data structure specifies an infinite object, often in the form of a tree.
- Notation: $\tau' = \mu X. \tau$ says “$\tau'$ is the infinite type defined by the equation $X = \tau$” Typically, $X$ occurs free in $\tau$.
- Example: lists
  - A list of elements of type $\tau$ (a $\tau$ list) is either empty or it is a pair of a $\tau$ and a $\tau$ list.
    
    $\tau$ list = $\text{Unit} + (\tau \times \tau$ list) or $\tau$ list = $\mu X. \text{Unit} + (\tau \times X)$
  - This is a recursive equation. We take its solution to be the smallest set of values $L$ that satisfies the equation
    
    $L = \text{nil} + (T \times L)$
  
    where $T$ is the set of values of type $\tau$. 

---

Friday, April 17, 2009
Formalization

- Two basic approaches, both address the question of the relationship between the type \( \mu X. \tau \) and its unfolding.
  - Equi-recursive: two types are definitionally equal, i.e., interchangeable in all contexts.
    - Both types represent the same infinite tree.
  - Iso-recursive: different but isomorphic.
    - Provide operations that “witness” the isomorphism.

- Equi-recursive formulation is intuitively cleaner, but more difficult to implement:
  - how does a typechecker manipulate infinite trees?
- Iso-recursive types notationally heavier, but pragmatic.
  - For example, datatypes in ML expressed using iso-recursive types.
Iso-recursive types

- Introduce a pair of functions for each recursive type:
  - $\text{unfold} : \mu X. \tau \rightarrow [\mu X. \tau/X] \tau$
  - $\text{fold} : [\mu X. \tau/X] \tau \rightarrow \mu X. \tau$

- The unfolding of a recursive type $\mu X. \tau$ is the type derived by replacing all occurrences of $X$ in $\tau$ by $\mu X. \tau$
Static Semantics of Recursive Types

- The typing rules are syntax directed
- Often, for syntactic simplicity, the fold and unfold operators are omitted
  - This makes type checking somewhat harder
Dynamics of Recursive Types

- We add a new form of value
  \[ v ::= ... \mid \text{fold}_{\mu \top \tau} v \]
  - The purpose of fold is to ensure that the value has the recursive type and not its unfolding

- The evaluation rules:
  \[
  \begin{align*}
  e \Downarrow v & \quad \text{fold}_{\mu \top \tau} e \Downarrow \text{fold}_{\mu \top \tau} v \\
  e \Downarrow \text{fold}_{\mu \top \tau} v & \quad \text{unfold}_{\mu \top \tau} e \Downarrow v
  \end{align*}
  \]
  - The folding annotations are for type checking only
  - They can be dropped after type checking
Proof Trees

Consider the term: \( \lambda x. (x \: x) \). What type can it be assigned?

- \((\lambda x. (x \: x)) \: (\lambda x. (x \: x))\)
- \((\lambda x. (x \: x)) \: (\lambda y. \text{false})\)

Former diverges, but latter has type \( \mu t. (t \rightarrow \text{bool}) \)

Written using fold:

\[
\text{fold}_{\mu t. t \rightarrow \text{bool}}(\lambda x: \mu t. t \rightarrow \text{bool}). ((\text{unfold} \: x) \: x)
\]
Proof Trees

\[
\begin{align*}
\{x : \mu \ t. \ t \rightarrow \text{bool}\} & \vdash x : (\mu \ t. \ t \rightarrow \text{bool}) \\
\{x : \mu \ t. \ t \rightarrow \text{bool}\} & \vdash \text{unfold } x : \{\mu \ t. \ t \rightarrow \text{bool}\} \rightarrow \text{bool} \ldots \\
\{x : \mu \ t. \ t \rightarrow \text{bool}\} & \vdash ((\text{unfold } x \cdot x) \cdot x) : \text{bool} \\
& \vdash [\lambda x : (\mu \ t. \ t \rightarrow \text{bool}).((\text{unfold } x \cdot x))]: (\mu \ t. \ t \rightarrow \text{bool}) \rightarrow \text{bool} \\
& \vdash [\text{fold}_{\mu \ t. \ t \rightarrow \text{bool}}(\lambda x : (\mu t. \ t \rightarrow \text{bool}).((\text{unfold } x \cdot x)))] : \mu \ t. \ t \rightarrow \text{bool}
\end{align*}
\]
Example with Recursive Types

- Lists
  \[ \tau \text{ list} = \mu t. (\text{Unit} + (\tau \times t)) \]
  \[ \text{nil}_\tau = \text{fold}_\tau \text{ list } (\text{injl unit}) \]
  \[ \text{cons}_\tau = \lambda x:\tau. \lambda l:\tau \text{ list}. \text{fold}_\tau \text{ list } \text{injr} (x, l) \]

- A list length function
  \[ \text{length}_\tau = \lambda l:\tau \text{ list}. \text{case } (\text{unfold}_\tau \text{ list } l) \text{ of} \]
  \[ \text{injl} \ x \Rightarrow 0 \]
  \[ | \text{injr} \ y \Rightarrow 1 + \text{length}_\tau (\text{snd} \ y) \]
More Examples

- \( \text{hd} = \lambda l : \tau \text{ list}. \text{case } (\text{unfold}_\tau \text{ list } l) \text{ of} \)
  \( \text{injl } x \Rightarrow \text{Error} \)
  \( | \text{injr } (a,b) \Rightarrow \ b \)

- \( \text{Hungry} = \mu \tau. \text{Nat} \rightarrow \tau \)
  \( f = \text{fix } \lambda f. \text{Nat} \rightarrow \text{Hungry} \)
  \( \lambda n : \text{Nat}. \text{fold}_{\text{Hungry}} f) : \)
  \( \mu \tau. \text{Nat} \rightarrow (\text{Nat} \rightarrow \tau) \equiv \text{Hungry} \)
  \( (\text{unfold } \ldots (\text{unfold } f ) 0) 1) 2) \ldots) : \text{Hungry} \)
More Examples

Integer streams: \( \text{stream} = \mu \tau. \text{Unit} \rightarrow (\text{int} \times \tau) \)

- **Destructors:**
  \[
  \text{hd} = \lambda s: \text{stream}. \quad \text{case } (\text{unfold}_{\text{stream}} s) \text{ unit of } \quad (h,r) \rightarrow h
  \]
  \[
  \text{tl} = \lambda s: \text{stream}. \quad \text{case } (\text{unfold}_{\text{stream}} s) \text{ unit of } \quad (h,r) \rightarrow r
  \]

- **Constructor:**
  \[
  \text{upfrom0} = (\text{fix } (\lambda f: \text{int} \rightarrow \text{stream}. \quad (\lambda n: \text{int}. \quad \text{fold}_{\text{stream}} (\lambda _: \text{Unit}. (n, f (n+1)))))) \ 0
  \]
  \[
  \text{hd upfrom0 } \rightarrow 0
  \]
  \[
  \text{hd } (\text{tl } (\text{tl } (\text{tl } \text{upfrom0}))) \rightarrow 3
  \]
Processes

- A process is a function that accepts some input and returns an output, along with another process:
  - process = μτ. Nat → (Nat x τ)
  
  p = fix (λf: Nat → process.
         λacc: Nat.
         foldprocess λn: Nat. let new = n + acc
             in (new, f acc)
         end)

  curr = λs: process. fst((unfoldprocess (s) 0))

  send = λn: Nat. λs: process. snd( unfoldprocess(s) n)

  send: Nat → process → process
Functional Objects

- Rather than maintaining internal (mutable) state, have operations on objects return new objects
  - `counter = μτ. \{get:Nat, inc: Unit → τ\}
  - objects are now recursively defined records
    
    ```
    c = let create = fix (λf. \{x:Nat\} → counter.
                        
                        λs: \{x:Nat\}.
                        λs: \{x:Nat\}.
                        fold_{counter}
                        
                        \{get = s.x,
                        \{get = s.x,
                        inc = λ_:Unit. f \{x = succ(s.x)\} \}
                        inc = λ_:Unit. f \{x = succ(s.x)\} \}
                          
                          in create \{x = 0\}
                          in create \{x = 0\}
    ```
Recursive Types in ML

- ML uses a simple syntactic trick to avoid having to write the explicit fold and unfold
- In ML recursive types are bundled with union types
  
  \[
  \text{datatype } t = C_1 \text{ of } \tau_1 | C_2 \text{ of } \tau_2 | \ldots | C_n \text{ of } \tau_n \text{ (} t \text{ can appear in } \tau_i) \]
  
  - E.g., datatype intlist = Nil of unit | Cons of int * intlist
- When the programmer writes
  \[
  \text{Cons (5, l)}
  \]
  
  the compiler treats it as
  \[
  \text{fold}_{\text{intlist}} (\text{injr (5, l)})
  \]
- When the programmer writes
  \[
  \text{case } e \text{ of Nil } \Rightarrow \ldots | \text{Cons (h, t) } \ldots
  \]
  
  the compiler treats it as
  \[
  \text{case unfold}_{\text{intlist}} e \text{ of Nil } \Rightarrow \ldots | \text{Cons (h,t) } \ldots
  \]
Encoding Call-by-Value $\lambda$-calculus

- So far, the simply typed $\lambda$ calculus was so weak that we could not encode non-terminating computations
  - Cannot encode recursion
  - Cannot write $\lambda x.x \ x$ (self-application)
  - Trick is to use a recursive type to type the two occurrences of the subexpression $x$.
    - $x$ must have arrow type whose domain is the same as the type of $x$ itself.
    - No finite type has this property, but $\mu \tau. \tau \rightarrow \tau$ does
- The addition of recursive types makes typed $\lambda$-calculus as expressive as untyped $\lambda$-calculus.

- We can show a conversion algorithm from call-by-value untyped $\lambda$-calculus to call-by-value typed $\lambda$ calculus equipped with recursive types.
Untyped Programming

- We write $\overline{e}$ for the conversion of the term $e$ to typed $\lambda$ calculus with recursive types ($F_{1^\mu}$)
  - The type of $\overline{e}$ is $V = \mu t. t \rightarrow t$
- The conversion rules
  - $\overline{x} = x$
  - $\overline{\lambda x. e} = \text{fold}_V (\lambda x:V. e)$
  - $\overline{e_1 e_2} = (\text{unfold}_V e_1) e_2$
- Verify that
  1. $\vdash \overline{e} : V$
  2. $e \Downarrow v$ if and only if $\overline{e} \Downarrow v$
Fixpoint operator

\[ \text{fix} = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

\( x \) needs to have an arrow type whose domain is the type of \( x \) itself satisfied by the recursive type \( \mu \tau'. \tau' \to \tau \)

A well-typed fixpoint operator:

\[ \text{fix}_\tau = \lambda f: \tau \to \tau. \]

\[ (\lambda x:\mu \tau'. \tau' \to \tau). f ((\text{unfold}_{\mu \tau'} \tau' \to \tau) x) x) \]

\[ (\text{fold}_{\mu \tau'} \tau' \to \tau (\lambda x:\mu \tau'. \tau' \to \tau. f ((\text{unfold}_{\mu \tau'} \tau' \to \tau) x) x))) \]
Subtyping Recursive Types

Under what conditions is: \( \mu X. \tau <: \mu X. \sigma \) ?

Can we simply use:

\[
\tau <: \sigma \\
\frac{\mu X. \tau <: \mu X. \sigma}{\mu X. \tau <: \mu X. \sigma}
\]
Subtyping Recursive Types

Consider the following:

- Suppose: $\tau <: \sigma$
- $r1: \mu X. \{ \text{nil} : \text{Unit}, \text{cons} : \sigma \times X \}$
- $r2: \mu X. \{ \text{nil} : \text{Unit}, \text{cons} : \tau \times X \}$
- Does the subtype assumption allow us to conclude that $r2 <: r1$?
Subtyping Recursive Types

\[
\text{Unit} <: \text{Unit} \quad \frac{\tau <: \sigma \quad X <: X}{(\tau \times X) <: (\sigma \times X)}
\]

\[
\{\text{nil} : \text{Unit}, \text{cons} : \langle\tau \times X\rangle\} <: \{\text{nil} : \text{Unit}, \text{cons} : \langle\sigma \times X\rangle\}
\]

\[r_2 <: r_1\]
Subtyping Recursive Types

Consider the following:

- Suppose: \( \tau <: \sigma \)
- \( r_1: \mu X. \{ \text{first}: \sigma, \text{rest}: X, \text{merge}: X \to X \} \)
- \( r_2: \mu X. \{ \text{first}: \tau, \text{rest}: X, \text{merge}: X \to X \} \)
- Does the subtype assumption allow us to conclude that \( r_2 <: r_1 \)?
Subtyping Recursive Types

\[
\begin{align*}
\tau & <: \sigma & r_2 & <: r_1 & (r_2 \rightarrow r_2) & <: (r_1 \rightarrow r_1) \\
\{\text{first} : \tau, \text{rest} : r_2, \text{merge} : r_2 \rightarrow r_2\} & <: \{\text{first} : \sigma, \text{rest} : r_1, \text{merge} : r_1 \rightarrow r_1\} \\
\hline \\
& r_2 <: r_1
\end{align*}
\]
Subtyping Recursive Types

Consider the following:

- Suppose: \( \tau <: \sigma \)
- \( r1: \mu X. \{ \text{first:} \sigma, \text{rest:} X, \text{merge:} X \rightarrow X \} \)
- \( r2: \mu X. \{ \text{first:} \tau, \text{rest:} X, \text{merge:} r1 \rightarrow X \} \)
- Does the subtype assumption allow us to conclude that \( r2 <: r1 \)?
New Rules

- fold and unfold operations

\[ \sigma <: [X \mapsto \mu X.t]t \quad [X \mapsto \mu X.t]t <: \tau \]
\[ \sigma <: \mu X.t \quad \mu X.t <: \tau \]

- Amber Rules

\[ \Sigma, X <: Y \vdash S <: T \]
\[ \Sigma \vdash \mu X.S <: \mu Y.T \]
\[ (X <: Y) \in \Sigma \]
\[ \Sigma \vdash X <: Y \]