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:
    $$\tau\text{ list} = \text{Unit} + (\tau \times \tau\text{ list})$$
    $$\tau\text{ list} = \mu X.\text{Unit} + (\tau \times X)$$
  - 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 $\tau$. 
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:
  - `unfold : μX.τ → [μX.τ/X]τ`
  - `fold : [μX.τ/X]τ → μX.τ`

- The unfolding of a recursive type μX.τ is the type derived by replacing all occurrences of X in τ by μX.τ
Static Semantics of Recursive Types

\[
\Gamma \vdash e : \mu t.\tau \\
\Gamma \vdash \text{unfold}_{\mu t.\tau} e : [\mu t.\tau / t]\tau
\]

\[
\Gamma \vdash e : [\mu t.\tau / t]\tau \\
\Gamma \vdash \text{fold}_{\mu t.\tau} e : \mu t.\tau
\]

- 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

\[ v ::= \ldots \mid \text{fold}_{\mu t.\tau} v \]

- The purpose of fold is to ensure that the value has the recursive type and not its unfolding

- The evaluation rules:

\[
\begin{align*}
\text{fold}_{\mu t.\tau} e \Downarrow v & \quad e \Downarrow v \\
\text{fold}_{\mu t.\tau} e \Downarrow \text{fold}_{\mu t.\tau} v & \quad e \Downarrow \text{fold}_{\mu t.\tau} v \\
\text{unfold}_{\mu t.\tau} e \Downarrow v & \quad \text{unfold}_{\mu t.\tau} e \Downarrow v
\end{align*}
\]

- 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

\( \mu t. (t \to \text{bool}) \)

Written using fold:

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

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

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

- 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

- \( \text{hd} = \lambda l: \tau \text{ list. case } (\text{unfold}_\tau \text{ list } l) \text{ of} \)
  
  \[
  \begin{align*}
  \text{injl} \ x & \Rightarrow \text{Error} \\
  \mid \text{injr} \ (a,b) & \Rightarrow \ b
  \end{align*}
  \]

- \( \text{Hungry} = \mu \tau. \text{Nat} \to \tau \)
  
  \[
  \begin{align*}
  f = \text{fix} \lambda f. \text{Nat} \to \text{Hungry}.
  \end{align*}
  \]

  \[
  \lambda n: \text{Nat}. \text{fold}_{\text{Hungry}} f): \\
  \mu \tau. \text{Nat} \to (\text{Nat} \to \tau) \equiv \text{Hungry}
  \]

  \[
  (\text{unfold } f) \ 0 \ 1 \ 2 \ \ldots \ : \text{Hungry}
  \]
More Examples

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

- **Destructors:**
  \[
  \text{hd} = \lambda s: \text{stream}. \left\langle \text{case (unfold_stream s) unit of } \begin{align*} \text{(h,r)} &\rightarrow h \\ \text{(h,r)} &\rightarrow r \end{align*} \right. \]
  \[
  \text{tl} = \lambda s: \text{stream}. \left\langle \text{case (unfold_stream s) unit of } \begin{align*} \text{(h,r)} &\rightarrow r \\ \text{(h,r)} &\rightarrow h \end{align*} \right. \]

- **Constructor:**
  \[
  \text{upfrom0} = \text{unfold_stream} \left( \text{fix} \left( \lambda f: \text{int} \rightarrow \text{stream}. \left( \lambda n: \text{int}. \text{fold_stream} (\lambda _: \text{Unit}. (n, f(n+1)))) \right) \right) \]

\[
\begin{align*}
\text{hd upfrom0} &\rightarrow 0 \\
\text{hd (tl (tl (tl upfrom0)))} &\rightarrow 3
\end{align*}
\]
Processes

- A process is a function that accepts some input and returns an output, along with another process:
  - process = μτ. Nat → (Nat × τ)
  - p = fix (λf: Nat → process.
      λacc: Nat.
      fold\text{process } λn: Nat. let new = n + acc
      in (new, f acc)
      end
  curr = λs: process. fst((unfold\text{process } (s) 0))
  send = λn:Nat.λs:process. snd( unfold\text{process}(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
  
  \[ c = \text{let create} = \text{fix } (\lambda f. \{x:\text{Nat}\} \rightarrow \text{counter.} \] 
  \[ \lambda s: \{x:\text{Nat}\}. \] 
  \[ \text{fold}_{\text{counter}} \] 
  \[ \{\text{get} = s.x, \] 
  \[ \text{inc} = \lambda _:\text{Unit}. f \{x = \text{succ}(s.x)\} \} \] 
  \[ \text{in create} \{x = 0\} \]
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.
  \[
  \text{datatype } t = C_1 \text{ of } \tau_1 \mid C_2 \text{ of } \tau_2 \mid \ldots \mid C_n \text{ of } \tau_n \text{ (} t \text{ can appear in } \tau_i \text{)}
  \]
  - 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 $e\downarrow$ for the conversion of the term $e$ to typed $\lambda$ calculus with recursive types ($F_{1}^{\mu}$)
  - The type of $e$ is $V = \mu t. t \to t$
- The conversion rules
  - $x = x$
  - $\lambda x. e = \text{fold}_V (\lambda x:V. e)$
  - $e_1 e_2 = (\text{unfold}_V e_1) e_2$
- Verify that
  1. $\vdash e : V$
  2. $e \Downarrow v$ if and only if $e\downarrow v$
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:

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

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

\[ (\text{fold}_{\mu \; \tau'. \tau'} \to \tau \; (\lambda x: \mu \; \tau'. \tau' \to \tau. \]

\[ f ((\text{unfold}_{\mu \; \tau'} \to \tau \; x) \; x))) \]
Subtyping Recursive Types

- Recall $\tau\text{ list} = \mu t. (\text{unit} + \tau \times t)$
  - We would like $\tau\text{ list} \ll \sigma\text{ list}$ whenever $\tau \ll \sigma$
- Try simple covariance:

$$
\frac{\tau \ll \sigma}{\mu t. \tau < \mu t. \sigma}
$$

Wrong!

This is wrong if $t$ occurs contravariantly in $\tau$

Take $\tau = t \rightarrow \text{int}$ and $\sigma = t \rightarrow \text{real}$

Above rule says that $\tau \ll \sigma$

We have $\tau \ll t \rightarrow \text{int}$ and $\sigma \ll \sigma \rightarrow \text{real}$

$\tau \ll \sigma$ would mean covariant function types
Subtyping Recursive Types (Cont.)

- The correct rule

\[
\begin{align*}
  t & < s \\
  \vdots \\
  \tau & < \sigma \\
  \mu t. \tau & < \mu s. \sigma
\end{align*}
\]

- We add as an assumption that the type variables stand for types with the desired subtype relationship
  - Before we assumed that they stood for the same type!

Verify that subtyping now works properly for lists
  - There is no subtyping between \( \mu t.t \rightarrow \text{int} \) and \( \mu t.t \rightarrow \text{real} \)
Foundations

- Recursive types characterize infinite objects.
- Our current proof techniques using induction and least-fixed point semantics is not suitable for describing these objects defined using an equi-recursive interpretation.
- Use the principle of co-induction and consider greatest fixed points of the sets characterized by these types.