Where are we

- System F gave us type abstraction
  - code reuse
  - strong abstractions
  - different from real languages (like ML), but the right foundation

- This lecture: Recursive Types (different use of type variables)
  - For building unbounded data structures
  - Turing-completeness without a fix primitive

- Future lecture (?): Existential types (dual to universal types)
  - First-class abstract types
  - Closely related to closures and objects

- Future lecture (?): Type-and-effect systems
Recursive Types

We could add list types (\(\text{list}(\tau)\)) and primitives (\([\ ], ::, \text{match}\)), but we want user-defined recursive types

Intuition:

\[
\text{type intlist} = \text{Empty} \mid \text{Cons int } * \text{ intlist}
\]

Which is roughly:

\[
\text{type intlist} = \text{unit} + (\text{int } * \text{ intlist})
\]

- Seems like a named type is unavoidable
  - But that’s what we thought with let rec and we used fix

- Analogously to \(\text{fix } \lambda x. e\), we’ll introduce \(\mu \alpha. \tau\)
  - Each \(\alpha\) “stands for” entire \(\mu \alpha. \tau\)
Mighty $\mu$

In $\tau$, type variable $\alpha$ stands for $\mu\alpha.\tau$, bound by $\mu$

Examples (of many possible encodings):

- int list (finite or infinite): $\mu\alpha.\text{unit} + (\text{int} \times \alpha)$
- int list (infinite “stream”): $\mu\alpha.\text{int} \times \alpha$
  - Need laziness (thunking) or mutation to build such a thing
  - Under CBV, can build values of type $\mu\alpha.\text{unit} \rightarrow (\text{int} \times \alpha)$
- int list list: $\mu\alpha.\text{unit} + ((\mu\beta.\text{unit} + (\text{int} \times \beta)) \times \alpha)$

Examples where type variables appear multiple times:

- int tree (data at nodes): $\mu\alpha.\text{unit} + (\text{int} \times \alpha \times \alpha)$
- int tree (data at leaves): $\mu\alpha.\text{int} + (\alpha \times \alpha)$
Using $\mu$ types

How do we build and use int lists ($\mu\alpha.\text{unit} + (\text{int} \times \alpha)$)?

We would like:

- empty list $= A(())$
  Has type: $\mu\alpha.\text{unit} + (\text{int} \times \alpha)$

- cons $= \lambda x:\text{int}. \lambda y:(\mu\alpha.\text{unit} + (\text{int} \times \alpha)). B((x, y))$
  Has type:
  $\text{int} \rightarrow (\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \rightarrow (\mu\alpha.\text{unit} + (\text{int} \times \alpha))$

- head $=$
  $\lambda x:(\mu\alpha.\text{unit} + (\text{int} \times \alpha)). \text{match } x \text{ with } A_. \ A(()) \mid B y. \ B(y.1)$
  Has type: $(\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \rightarrow (\text{unit} + \text{int})$

- tail $=$
  $\lambda x:(\mu\alpha.\text{unit} + (\text{int} \times \alpha)). \text{match } x \text{ with } A_. \ A(()) \mid B y. \ B(y.2)$
  Has type:
  $(\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \rightarrow (\text{unit} + \mu\alpha.\text{unit} + (\text{int} \times \alpha))$

But our typing rules allow none of this (yet)
Using $\mu$ types (continued)

For empty list = $A(())$, one typing rule applies:

$$
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2
$$

So we could show

$$
\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2
$$

But we want $\mu\alpha.\text{unit} + (\text{int} \ast \alpha)$

Notice: $\text{unit} + (\text{int} \ast (\mu\alpha.\text{unit} + (\text{int} \ast \alpha)))$ is

$(\text{unit} + (\text{int} \ast \alpha))[((\mu\alpha.\text{unit} + (\text{int} \ast \alpha))/\alpha]$

The key: Subsumption — recursive types are equal to their “unfolding” or “unfolding” (equi-recursive).
Return of subtyping

Can use \( \Gamma \vdash e : \tau \) and these subtyping rules:

- **Fold**
  \[
  \tau[\langle \mu \alpha. \tau \rangle / \alpha] \leq \mu \alpha. \tau
  \]

- **Unfold**
  \[
  \mu \alpha. \tau \leq \tau[\langle \mu \alpha. \tau \rangle / \alpha]
  \]

Subtyping can “fold” or “unfold” a recursive type

```
unfold[\mu \alpha. \tau]  \
\mu \alpha. \tau  \
\tau[\langle \mu \alpha. \tau \rangle / \alpha]  \\fold[\mu \alpha. \tau]
```
Folding and unfolding (cont.)

The fold and unfold maps are provided as primitives by the language.

Can now give empty-list, cons, and head the types we want: Constructors use fold, destructors use unfold

Notice how little we did: One new form of type \((\mu \alpha. \tau)\) and two new subtyping rules.
Metatheory

What is the relation between the type $\mu \alpha. \tau$ and its one-step unfolding?

- Equi-recursive (implicit) approach (subsumption): takes a recursive type and its unfolding as definitionally equal – interchangeable in all contexts (it’s the type checker’s responsibility to make sure that a term of one type will be allowed as an argument to a function expecting the other). Example: [http://whiley.org/2011/02/16/minimising-recursive-data-types/](http://whiley.org/2011/02/16/minimising-recursive-data-types/).

- Iso-recursive (explicit) approach: takes a recursive type and its unfolding as different, but isomorphic.

Boyana Norris
CIS 624 2014, Lecture 18
Despite additions being minimal, must reconsider how recursive types change STLC and System F:

- Erasure (no run-time effect): unchanged

- Termination: changed!
  - \((\lambda x: \mu \alpha. \alpha \to \alpha. \ x\ x)(\lambda x: \mu \alpha. \alpha \to \alpha. \ x\ x)\)
  - In fact, we’re now Turing-complete without fix (actually, can type-check every closed \(\lambda\) term)

- Safety: still safe, but Canonical Forms harder

- Inference: Shockingly efficient for “STLC plus \(\mu\)” (A great contribution of PL theory with applications in OO and XML-processing languages)
Syntax-directed $\mu$ types

(Equi-recursive) recursive types via subsumption “seem magical”

Instead, we can make programmers tell the type-checker where/how to fold and unfold

“Iso-recursive” types: remove subtyping and add expressions:

\[
\begin{align*}
\tau & ::= \ldots | \mu \alpha. \tau \\
\mathit{e} & ::= \ldots | \mathit{fold}_{\mu \alpha. \tau} \ e \ | \ \mathit{unfold} \ e \\
\mathit{v} & ::= \ldots | \mathit{fold}_{\mu \alpha. \tau} \ \mathit{v} \\
\end{align*}
\]

\[
\begin{align*}
\mathit{e} & \rightarrow \mathit{e}' \\
\mathit{fold}_{\mu \alpha. \tau} \ e & \rightarrow \mathit{fold}_{\mu \alpha. \tau} \ \mathit{e}' \\
\mathit{e} & \rightarrow \mathit{e}' \\
\mathit{unfold} \ e & \rightarrow \mathit{unfold} \ \mathit{e}' \\
\mathit{unfold} \ (\mathit{fold}_{\mu \alpha. \tau} \ \mathit{v}) & \rightarrow \mathit{v}
\end{align*}
\]

\[
\begin{align*}
\Delta; \Gamma & \vdash \mathit{e} : \tau[(\mu \alpha. \tau)/\alpha] \\
\Delta; \Gamma & \vdash \mathit{fold}_{\mu \alpha. \tau} \ e : \mu \alpha. \tau \\
\Delta; \Gamma & \vdash \mathit{unfold} \ e : \tau[(\mu \alpha. \tau)/\alpha]
\end{align*}
\]
Syntax-directed, continued

Type-checking is syntax-directed / No subtyping necessary

Canonical Forms, Preservation, and Progress are simpler

This is an example of a key trade-off in language design:
- Implicit typing can be impossible, difficult, or confusing
- Explicit coercions can be annoying and clutter language with no-ops
- Most languages do some of each

Anything is decidable if you make the code producer give the implementation enough “hints” about the “proof”
ML datatypes revealed

How is $\mu\alpha.\tau$ related to type $t = \text{Foo of int} \mid \text{Bar of int * t}$

Constructor use is a “sum-injection” followed by an *implicit fold*

▶ So $\text{Foo } e$ is really $\text{fold}_t \text{ Foo}(e)$
▶ That is, $\text{Foo } e$ has type $t$ (the folded type)

A pattern-match has an *implicit unfold*

▶ So match $e$ with... is really match $\text{unfold } e$ with...

This “trick” works because different recursive types use different tags – so the type-checker knows *which* type to fold to