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

Recursive Types

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

Intuition:

type intlist = Empty | Cons int * intlist

Which is roughly:

type intlist = unit + (int * intlist)

- Seems like a named type is unavoidable
  - But that's what we thought with \texttt{let rec} and we used \texttt{fix}

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

Examples where type variables appear multiple times:

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

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

We would like:
- empty list = \( A(()) \)
  Has type: \( \ldots \rightarrow (\text{unit} + \mu \alpha. \text{unit} + (\text{int} \ast \alpha)) \)
- cons = \( \lambda x: \text{int}. \lambda y: (\mu \alpha. \text{unit} + (\text{int} \ast \alpha)). B((x, y)) \)
  Has type:
  \[
  \text{int} \rightarrow (\mu \alpha. \text{unit} + (\text{int} \ast \alpha)) \rightarrow (\mu \alpha. \text{unit} + (\text{int} \ast \alpha))
  \]
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:
  \( \text{int} \to (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \to (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \)
- head =
  \[ \lambda x:(\mu \alpha. \text{unit} + (\text{int} \times \alpha)). \text{match } x \text{ with } A.y. B(y.1) \]
  Has type:
  \( (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \text{int}) \)
- tail =
  \[ \lambda x:(\mu \alpha. \text{unit} + (\text{int} \times \alpha)). \text{match } x \text{ with } A.y. B(y.2) \]
  Has type:
  \( (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \mu \alpha. \text{unit} + (\text{int} \times \alpha)) \)

But our typing rules allow none of this (yet)

Using \( \mu \) types (continued)

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

\[
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2 \\
\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2
\]

So we could show
\[
\Delta; \Gamma \vdash A(() : \text{unit} + (\text{int} \times (\mu \alpha. \text{unit} + (\text{int} \times \alpha)))
\]
(since \( FTV(\text{int} \times \mu \alpha. \text{unit} + (\text{int} \times \alpha)) = \emptyset \subseteq \Delta \))

Using \( \mu \) types (continued)

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

\[
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2 \\
\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2
\]

So we could show
\[
\Delta; \Gamma \vdash A(() : \text{unit} + (\text{int} \times (\mu \alpha. \text{unit} + (\text{int} \times \alpha)))
\]
(since \( FTV(\text{int} \times \mu \alpha. \text{unit} + (\text{int} \times \alpha)) = \emptyset \subseteq \Delta \))

But we want \( \mu \alpha. \text{unit} + (\text{int} \times \alpha) \)
Using \( \mu \) types (continued)

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

\[
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2
\]
\[
\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2
\]

So we could show
\[
\Delta; \Gamma \vdash A(() : \text{unit} + (\text{int} \ast (\mu\alpha.\text{unit} + (\text{int} \ast \alpha))))
\]

(since \( FTV(\text{int} \ast (\mu\alpha.\text{unit} + (\text{int} \ast \alpha))) = \emptyset \subseteq \Delta \))

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

Return of subtyping

Can use subsumption and these subtyping rules:

**Roll**

\[
\tau[(\mu\alpha.\tau)/\alpha] \leq \mu\alpha.\tau
\]

**Unroll**

\[
\mu\alpha.\tau \leq \tau[(\mu\alpha.\tau)/\alpha]
\]

Subtyping can “roll” or “unroll” a recursive type

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

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

(Skipping: Depth subtyping on recursive types is very interesting)

Metatheory

Despite additions being minimal, must reconsider how recursive types change STLC and System F:

▶ Erasure (no run-time effect): unchanged

▶ Termination: changed!
  ▶ \((\lambda x:\mu\alpha.\alpha \rightarrow \alpha.\ x\ x)(\lambda x:\mu\alpha.\alpha \rightarrow \alpha.\ x\ x)\)
  ▶ 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

Recursive types via subsumption “seems magical”

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

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

$$
\begin{align*}
\tau & ::= \cdots \mid \mu \alpha. \tau \\
e & ::= \cdots \mid \text{roll}_{\mu \alpha. \tau} e \mid \text{unroll} e \\
v & ::= \cdots \mid \text{roll}_{\mu \alpha. \tau} v
\end{align*}
$$

$$
\frac{e \rightarrow e'}{\text{roll}_{\mu \alpha. \tau} e \rightarrow \text{roll}_{\mu \alpha. \tau} e'}
\quad
\frac{e \rightarrow e'}{\text{unroll} e \rightarrow \text{unroll} e'}
$$

$$
\Delta; \Gamma \vdash e : \tau[(\mu \alpha. \tau)/\alpha]
\quad
\Delta; \Gamma \vdash e : \mu \alpha. \tau
\quad
\Delta; \Gamma \vdash \text{unroll} e : \tau[(\mu \alpha. \tau)/\alpha]
$$

ML datatypes revealed

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

Constructor use is a “sum-injection” followed by an implicit $\text{roll}$

- So $\text{Foo } e$ is really $\text{roll} \text{ Foo}(e)$
- That is, $\text{Foo } e$ has type $t$ (the rolled type)

A pattern-match has an implicit $\text{unroll}$

- So match $e$ with... is really match $\text{unroll} e$ with...

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

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”