CSE 505: Programming Languages

Lecture 17 — Recursive Types

Zach Tatlock
Autumn 2015
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 (list(\(\tau\))) and primitives (\([]\), ::, match), but we want user-defined recursive types.

Intuition:

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

Which is roughly:

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

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

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

Examples where type variables appear multiple times:
- int tree (data at nodes): $\mu\alpha.\text{unit} + (\text{int} \ast \alpha \ast \alpha)$
- int tree (data at leaves): $\mu\alpha.\text{int} + (\alpha \ast \alpha)$
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 = $\lambda x:(\mu_\alpha.\text{unit} + (\text{int} \times \alpha))$. match $x$ with $A$. $A()$ | $B$. $y.1$
  Has type: $(\mu_\alpha.\text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \mu_\alpha.\text{unit} + (\text{int} \times \alpha))$

▶ tail = $\lambda x:(\mu_\alpha.\text{unit} + (\text{int} \times \alpha))$. match $x$ with $A$. $A()$ | $B$. $y.2$
  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

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

We would like:
- empty list $= A(())$
  Has type: $\mu\alpha.\text{unit} + (\text{int} \times \alpha)$
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: $\mu\alpha.\text{unit} + (\text{int} \ast \alpha)$
- 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** =
  $$\lambda x: (\mu\alpha.\text{unit} + (\text{int} \times \alpha)). \text{match } x \text{ with } A_. A(()) \mid B y. B(y.1)$$
  Has type: $(\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \text{int})$
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:
  $(\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \rightarrow (\text{unit} + \mu\alpha.\text{unit} + (\text{int} \times \alpha))$
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: $\mu \alpha. \text{unit} + (\text{int} \ast \alpha)$

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

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

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

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

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

$$
\begin{align*}
\Delta; \Gamma \vdash e : \tau_1 & \quad \Delta \vdash \tau_2 \\
\frac{}{\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2}
\end{align*}
$$

So we could show

$$
\Delta; \Gamma \vdash A(()) : \text{unit} + (\text{int} \ast (\mu\alpha.\text{unit} + (\text{int} \ast \alpha)))
$$

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

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2}$$

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

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

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

$$
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2 \\
\frac{}{\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2}
$$

So we could show

$$
\Delta; \Gamma \vdash A(()) : \text{unit} + (\text{int} \ast (\mu\alpha.\text{unit} + (\text{int} \ast \alpha)))
$$

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

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]

The key: Subsumption — recursive types are equal to their “unrolling”
Return of subtyping

Can use *subsumption* and these subtyping rules:

\[
\text{ROLL} \quad \tau[(\mu \alpha. \tau)/\alpha] \leq \mu \alpha. \tau
\]

\[
\text{UNROLL} \quad \mu \alpha. \tau \leq \tau[(\mu \alpha. \tau)/\alpha]
\]

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:

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

\[
\begin{array}{c}
\frac{e \rightarrow e'}{\text{roll}_{\mu \alpha.\tau} e \rightarrow \text{roll}_{\mu \alpha.\tau} e'} \\
\frac{e \rightarrow e'}{\text{unroll} e \rightarrow \text{unroll} e'} \\
\frac{}{\text{unroll} \left( \text{roll}_{\mu \alpha.\tau} v \right) \rightarrow v}
\end{array}
\]

\[
\begin{array}{c}
\frac{\Delta; \Gamma \vdash e : \tau[(\mu \alpha.\tau)/\alpha]}{\Delta; \Gamma \vdash \text{roll}_{\mu \alpha.\tau} e : \mu \alpha.\tau} \\
\frac{}{\Delta; \Gamma \vdash \text{unroll} e : \tau[(\mu \alpha.\tau)/\alpha]}
\end{array}
\]
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 roll*

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

A pattern-match has an *implicit unroll*

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

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