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(τ)) 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 let rec and we used fix
- Analogously to **fix** $\lambda x.\ e$, we'll introduce $\mu\alpha.\tau$
  - Each α "stands for" entire $\mu\alpha.\tau$

Mighty μ

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

Examples (of many possible encodings):

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

Examples where type variables appear multiple times:

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

Using μ types

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

We would like:

Using μ types

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

We would like:

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

Using μ types

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

We would like:

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

Using μ types

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

We would like:

- empty list = $\textbf{A}(())$
  Has type: $\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)$
- cons = $\lambda x{:}\textbf{int}.\ \lambda y{:}(\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)).\ \textbf{B}((x, y))$
  Has type:
  $\textbf{int} \rightarrow (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)) \rightarrow (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha))$
- head =
  $\lambda x{:}(\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)).\ \textbf{match}\ x\ \textbf{with}\ \textbf{A}\_.\ \textbf{A}(())\ |\ \textbf{B}y.\ \textbf{B}(y.1)$
  Has type: $(\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)) \rightarrow (\textbf{unit} + \textbf{int})$

Using μ types

How do we build and use int lists ($\mu\alpha$.**unit** + (**int** $*$ α))?

We would like:

- empty list = **A**(())
  Has type: $\mu\alpha$.**unit** + (**int** $*$ α)
- cons = λx:**int**. λy:($\mu\alpha$.**unit** + (**int** $*$ α)). **B**((x, y))
  Has type:
  int $\to$ ($\mu\alpha$.**unit** + (**int** $*$ α)) $\to$ ($\mu\alpha$.**unit** + (**int** $*$ α))
- head =
  λx:($\mu\alpha$.**unit** + (**int** $*$ α)). **match** x **with A**_. **A**(()) | **B**y. **B**(y.1)
  Has type: ($\mu\alpha$.**unit** + (**int** $*$ α)) $\to$ (**unit** + **int**)
- tail =
  λx:($\mu\alpha$.**unit** + (**int** $*$ α)). **match** x **with A**_. **A**(()) | **B**y. **B**(y.2)
  Has type:
  ($\mu\alpha$.**unit** + (**int** $*$ α)) $\to$ (**unit** + $\mu\alpha$.**unit** + (**int** $*$ α))

Using μ types

How do we build and use int lists ($\mu\alpha$.**unit** + (**int** $*$ α))?

We would like:

- empty list = **A**(())
  Has type: $\mu\alpha$.**unit** + (**int** $*$ α)
- cons = λx:**int**. λy:($\mu\alpha$.**unit** + (**int** $*$ α)). **B**((x, y))
  Has type:
  int $\to$ ($\mu\alpha$.**unit** + (**int** $*$ α)) $\to$ ($\mu\alpha$.**unit** + (**int** $*$ α))
- head =
  λx:($\mu\alpha$.**unit** + (**int** $*$ α)). **match** x **with A**_. **A**(()) | **B**y. **B**(y.1)
  Has type: ($\mu\alpha$.**unit** + (**int** $*$ α)) $\to$ (**unit** + **int**)
- tail =
  λx:($\mu\alpha$.**unit** + (**int** $*$ α)). **match** x **with A**_. **A**(()) | **B**y. **B**(y.2)
  Has type:
  ($\mu\alpha$.**unit** + (**int** $*$ α)) $\to$ (**unit** + $\mu\alpha$.**unit** + (**int** $*$ α))

But our typing rules allow none of this (yet)

Using μ types (continued)

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

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

So we could show
$\Delta;\Gamma \vdash$ **A**(()) : **unit** + (**int** $*$ ($\mu\alpha$.**unit** + (**int** $*$ α)))
(since FTV(**int** $*$ $\mu\alpha$.**unit** + (**int** $*$ α)) $= \emptyset \subseteq \Delta$)

Using μ types (continued)

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

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

So we could show
$\Delta;\Gamma \vdash$ **A**(()) : **unit** + (**int** $*$ ($\mu\alpha$.**unit** + (**int** $*$ α)))
(since FTV(**int** $*$ $\mu\alpha$.**unit** + (**int** $*$ α)) $= \emptyset \subseteq \Delta$)

But we want $\mu\alpha$.**unit** + (**int** $*$ α)

Using μ types (continued)

For empty list $= \mathsf{A}(())$, one typing rule applies:

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

So we could show
$\Delta;\Gamma \vdash \mathsf{A}(()) : \textbf{unit} + (\textbf{int} * (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)))$
(since $FTV(\textbf{int} * \mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)) = \emptyset \subseteq \Delta$)

But we want $\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)$

Notice: $\textbf{unit} + (\textbf{int} * (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)))$ is
$(\textbf{unit} + (\textbf{int} * \alpha))[(\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha))/\alpha]$

Using μ types (continued)

For empty list $= \mathsf{A}(())$, one typing rule applies:

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

So we could show
$\Delta;\Gamma \vdash \mathsf{A}(()) : \textbf{unit} + (\textbf{int} * (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)))$
(since $FTV(\textbf{int} * \mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)) = \emptyset \subseteq \Delta$)

But we want $\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)$

Notice: $\textbf{unit} + (\textbf{int} * (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)))$ is
$(\textbf{unit} + (\textbf{int} * \alpha))[(\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha))/\alpha]$

The key: Subsumption — recursive types are equal to their "unrolling"

Return of subtyping

Can use *subsumption* and these subtyping rules:

$$\textsc{Roll}\quad \frac{}{\tau[(\mu\alpha.\tau)/\alpha] \leq \mu\alpha.\tau} \qquad\qquad \textsc{Unroll}\quad \frac{}{\mu\alpha.\tau \leq \tau[(\mu\alpha.\tau)/\alpha]}$$

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!
  - $(\lambda x{:}\mu\alpha.\alpha \to \alpha.\ x\ x)(\lambda x{:}\mu\alpha.\alpha \to \alpha.\ x\ x)$
  - In fact, we're now Turing-complete without fix (actually, can type-check every closed λ term)
- Safety: still safe, but Canonical Forms harder
- Inference: Shockingly efficient for "STLC plus μ"
  (A great contribution of PL theory with applications in OO and XML-processing languages)

Syntax-directed μ 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:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mu\alpha.\tau \\ e & ::= & \ldots \mid \mathbf{roll}_{\mu\alpha.\tau}\ e \mid \mathbf{unroll}\ e \\ v & ::= & \ldots \mid \mathbf{roll}_{\mu\alpha.\tau}\ v \end{array}$$

$$\frac{e \to e'}{\mathbf{roll}_{\mu\alpha.\tau}\ e \to \mathbf{roll}_{\mu\alpha.\tau}\ e'} \qquad \frac{e \to e'}{\mathbf{unroll}\ e \to \mathbf{unroll}\ e'}$$

$$\frac{}{\mathbf{unroll}\ (\mathbf{roll}_{\mu\alpha.\tau}\ v) \to v}$$

$$\frac{\Delta;\Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha]}{\Delta;\Gamma \vdash \mathbf{roll}_{\mu\alpha.\tau}\ e : \mu\alpha.\tau} \qquad \frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau}{\Delta;\Gamma \vdash \mathbf{unroll}\ e : \tau[(\mu\alpha.\tau)/\alpha]}$$

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 = Foo of int | Bar of int * t`

Constructor use is a "sum-injection" followed by an *implicit roll*

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

A pattern-match has an *implicit unroll*

- So `match` e `with...` is really `match` **unroll** e `with...`

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