CSE-505: Programming Languages

Lecture 20.5 — Recursive Types

Zach Tatlock
2016

Recursive Types

We could add list types (list(τ)) 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 fix λx. 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): \(\mu \alpha. \text{int} + (\alpha \ast \alpha)\)

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
Using µ types

How do we build and use int lists (µα.unit + (int ∗ α))?  

We would like:

▶ empty list = A(())
  Has type: ... → (unit + µα.unit + (int ∗ α))

But our typing rules allow none of this (yet)

Zach Tatlock CSE-505 2016, Lecture 20.5 5

Using µ types

How do we build and use int lists (µα.unit + (int ∗ α))?  

We would like:

▶ empty list = A(())
  Has type: µα.unit + (int ∗ α)

Zach Tatlock CSE-505 2016, Lecture 20.5 5
Using μ types

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

We would like:
- empty list = \(A(())\)
  - Has type: \(μα.\text{unit} + (\text{int} \ast α)\)
- cons = \(λx:\text{int}. λy:(μα.\text{unit} + (\text{int} \ast α)). B((x, y))\)
  - Has type:
    \(\text{int} \to (μα.\text{unit} + (\text{int} \ast α)) \to (μα.\text{unit} + (\text{int} \ast α))\)
- head =
  \(λx:(μα.\text{unit} + (\text{int} \ast α)). \text{match } x \text{ with } A_. \ A(() \mid By. B(1))\)
  - Has type: \((μα.\text{unit} + (\text{int} \ast α)) \to (\text{unit} + \text{int})\)
- tail =
  \(λx:(μα.\text{unit} + (\text{int} \ast α)). \text{match } x \text{ with } A_. \ A(() \mid By. B(2))\)
  - Has type:
    \((μα.\text{unit} + (\text{int} \ast α)) \to (\text{unit} + μα.\text{unit} + (\text{int} \ast α))\)

But our typing rules allow none of this (yet)

Using μ types (continued)

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

\[ \Delta; \Gamma ⊢ e : τ₁ \quad Δ ⊢ τ₂ \]
\[ Δ; Γ ⊢ A(e) : τ₁ + τ₂ \]

So we could show
\[ Δ; Γ ⊢ A(() : \text{unit} + (\text{int} \ast (μα.\text{unit} + (\text{int} \ast α))))\]
(since \(FTV(\text{int} \ast μα.\text{unit} + (\text{int} \ast α)) = \emptyset \subseteq Δ\))

Using μ types

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

We would like:
- empty list = \(A(())\)
  - Has type: \(μα.\text{unit} + (\text{int} \ast α)\)
- cons = \(λx:\text{int}. λy:(μα.\text{unit} + (\text{int} \ast α)). B((x, y))\)
  - Has type:
    \(\text{int} \to (μα.\text{unit} + (\text{int} \ast α)) \to (μα.\text{unit} + (\text{int} \ast α))\)
- head =
  \(λx:(μα.\text{unit} + (\text{int} \ast α)). \text{match } x \text{ with } A_. \ A(() \mid By. B(1))\)
  - Has type: \((μα.\text{unit} + (\text{int} \ast α)) \to (\text{unit} + \text{int})\)
- tail =
  \(λx:(μα.\text{unit} + (\text{int} \ast α)). \text{match } x \text{ with } A_. \ A(() \mid By. B(2))\)
  - Has type:
    \((μα.\text{unit} + (\text{int} \ast α)) \to (\text{unit} + μα.\text{unit} + (\text{int} \ast α))\)

But we want \(μα.\text{unit} + (\text{int} \ast α)\)
Using μ 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
\[ \Delta; \Gamma \vdash A(() : \text{unit} + (\text{int} * (\mu \alpha. \text{unit} + (\text{int} * \alpha)))) \]

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

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

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

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

Return of subtyping

Can use subsumption and these subtyping rules:

\[
\begin{align*}
\text{ROLL} & \quad \text{UNROLL} \\
\tau[(\mu \alpha. \tau)/\alpha] & \leq \mu \alpha. \tau & \mu \alpha. \tau & \leq \tau[(\mu \alpha. \tau)/\alpha]
\end{align*}
\]

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!
  - \( (\lambda x: \mu \alpha. \alpha \rightarrow \alpha \cdot x \cdot x)(\lambda x: \mu \alpha. \alpha \rightarrow \alpha \cdot x \cdot x) \)
  - 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:

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

\[
\begin{align*}
e & \rightarrow e' \\
\text{roll}_{\mu \alpha.\tau} e & \rightarrow \text{roll}_{\mu \alpha.\tau} e' \\
\text{unroll} e & \rightarrow \text{unroll} e'
\end{align*}
\]

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

ML datatypes revealed

How is \( \mu \alpha.\tau \) related to

\[
t = \text{Foo of int} \mid \text{Bar of int * t}
\]

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

- So Foo \( e \) is really \( \text{roll}_{\mu \alpha.\tau} \text{Foo}(e) \)
- That is, Foo \( e \) has type \( t \) (the rolled type)

A pattern-match has an implicit unroll

- So match \( e \) with... is really match \( \text{unroll} e \) 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”