CIS 624: Structure of Programming Languages

Lecture 18 — Recursive Types

Boyana Norris
2013
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:

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

Which is roughly:

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

▶ 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 \( \muα.τ \)
  ▶ Each \( α \) “stands for” entire \( \muα.τ \)
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** =
  $\lambda x:(\mu\alpha.\text{unit} + (\text{int} \times \alpha)). \text{match } x \text{ with } A_. A(() \mid B y. B(y.1)$
  Has type: $(\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \text{int})$
- **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:
  $(\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \mu\alpha.\text{unit} + (\text{int} \times \alpha))$

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

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

(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

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

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

Can use *subsumption* and these subtyping rules:

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

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

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

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

(Skipping: Depth subtyping on recursive types)
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.
Metatheory (cont.)

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

- **Erasure (no run-time effect):** unchanged

- **Termination:** changed!
  - 
  
  
  
  - 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 $\mu$ 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:

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

$$
\begin{align*}
e \rightarrow e' & \quad \Rightarrow \\
\text{fold}_{\mu\alpha.\tau} e \rightarrow \text{fold}_{\mu\alpha.\tau} e' & \\
\text{unfold} e \rightarrow \text{unfold} e' & \\
\text{unfold} (\text{fold}_{\mu\alpha.\tau} v) & \rightarrow v
\end{align*}
$$

$$
\begin{align*}
\Delta; \Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha] & \quad \Rightarrow \\
\Delta; \Gamma \vdash \text{fold}_{\mu\alpha.\tau} e : \mu\alpha.\tau & \\
\Delta; \Gamma \vdash \text{unfold} e : \tau[(\mu\alpha.\tau)/\alpha] & \\
\Delta; \Gamma \vdash e : \mu\alpha.\tau
\end{align*}
$$
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”
How is $\mu \alpha. \tau$ related to type $t = \text{Foo of int} \mid \text{Bar of int} \times t$

Constructor use is a “sum-injection” followed by an \textit{implicit fold}

$\Rightarrow$ So $\text{Foo } e$ is really $\text{fold}_t \text{Foo}(e)$

$\Rightarrow$ That is, $\text{Foo } e$ has type $t$ (the folded type)

A pattern-match has an \textit{implicit unfold}

$\Rightarrow$ 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 \textit{which} type to fold to