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 \( \lambda x. e \), 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"): \( \mu \alpha. \text{int} + \alpha \ast \alpha \)
  - 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: \( \mu \alpha. \text{unit} + (((\mu \beta. \text{unit} + (\text{int} \ast \beta)) \ast \alpha) \ast \alpha) \)

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 (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

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

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

Can use subsumption and these subtyping rules:

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

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/
- Iso-recursive (explicit) approach: takes a recursive type and its unfolding as different, but isomorphic.

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:

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

\[
\begin{align*}
e \rightarrow e' & \quad \text{fold}_{\mu\alpha.\tau} e \rightarrow \text{fold}_{\mu\alpha.\tau} e' \\
\text{unfold} e \rightarrow \text{unfold} e' & \quad e \rightarrow e'
\end{align*}
\]

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

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 fold

- So Foo \(e\) is really fold Foo(e)
- That is, Foo \(e\) has type \(t\) (the folded type)

A pattern-match has an implicit unfold

- So match \(e\) with... is really match unfold \(e\) with...

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