CIS 624: Structure of Programming Languages

Lecture 18 — Recursive Types

Boyana Norris
2015

Recursive Types

We could add list types (\texttt{list(\tau)}) and primitives ([], ::, match), but we want user-defined recursive types.

Intuition:
\texttt{type intlist = Empty | Cons int * intlist}

Which is roughly:
\texttt{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 \texttt{fix \lambda x. e}, we'll introduce \texttt{\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):
- \texttt{int list} (finite or infinite): \mu \alpha.\texttt{unit} + (int * \alpha)
- \texttt{int list} (infinite "stream"): \mu \alpha.\texttt{int} + \alpha
  - Need laziness (thunking) or mutation to build such a thing
  - Under CBV, can build values of type \mu \alpha.\texttt{unit} \rightarrow (int * \alpha)
- \texttt{int list list}: \mu \alpha.\texttt{unit} + ((\mu \beta.\texttt{unit} + (int * \beta)) * \alpha)

Examples where type variables appear multiple times:
- \texttt{int tree} (data at nodes): \mu \alpha.\texttt{unit} + (int * \alpha * \alpha)
- \texttt{int tree} (data at leaves): \mu \alpha.\texttt{int} + (\alpha * \alpha)

Using \mu types

How do we build and use int lists (\mu \alpha.\texttt{unit} + (int * \alpha))? We would like:

- empty list = \texttt{A(())}
  - Has type: \mu \alpha.\texttt{unit} + (int * \alpha)
- \texttt{cons} = \lambda x::int. \lambda y:(\mu \alpha.\texttt{unit} + (int * \alpha)). \texttt{B}((x, y))
  - Has type: \texttt{int} \rightarrow (\mu \alpha.\texttt{unit} + (int * \alpha)) \rightarrow (\mu \alpha.\texttt{unit} + (int * \alpha))
- head = \lambda x:(\mu \alpha.\texttt{unit} + (int * \alpha)). \texttt{match} x \texttt{with} A\_ \_ A(()) | By. \texttt{B}(y,1)
  - Has type: (\mu \alpha.\texttt{unit} + (int * \alpha)) \rightarrow (unit + int)
- tail = \lambda x:(\mu \alpha.\texttt{unit} + (int * \alpha)). \texttt{match} x \texttt{with} A\_ \_ A(()) | By. \texttt{B}(y,2)
  - Has type: (\mu \alpha.\texttt{unit} + (int * \alpha)) \rightarrow (unit + \mu \alpha.\texttt{unit} + (int * \alpha))

But our typing rules allow none of this (yet)

Using \mu types (continued)

For empty list = \texttt{A(())}, one typing rule applies:
\[
\Delta ; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2
\]
\[
\Delta ; \Gamma \vdash \texttt{A}(e) : \tau_1 + \tau_2
\]

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

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

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

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

\[
\begin{align*}
\text{SUBSUMPTION} & : \quad \Gamma \vdash e : \tau' \quad \tau' \leq \tau \\
\text{FOLD} & : \quad \tau \leq \mu \alpha.\tau \quad \mu \alpha.\tau 
\end{align*}
\]

and these subtyping rules:

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

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

\[
\begin{array}{c}
\text{unfold}[\mu \alpha.\tau] \\
\mu \alpha.\tau \\
\tau[(\mu \alpha.\tau)/\alpha] \\
\text{fold}[\mu \alpha.\tau]
\end{array}
\]

Folding and unfolding (cont.)

The fold and unfold maps are provided as primitives by the language.

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.

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

( 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:

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

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

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

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”
ML datatypes revealed

How is $\mu\alpha.\tau$ related to

type $\tau$ = Foo of int | Bar of int * $\tau$

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 $\tau$ (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