Recursive Types

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

Intuition:

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

Which is roughly:

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

- Seems like a named type is unavoidable
  - But that’s what we thought with let rec and we used fix

- Analogously to \(\text{fix } \lambda x. e\), we’ll introduce \(\mu \alpha. \tau\)
  - Each \(\alpha\) “stands for” entire \(\mu \alpha. \tau\)

Using \(\mu\) types

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

We would like:

- empty list \(= A(())\)
  - Has type: \(\mu \alpha. \text{unit } + (\text{int } * \alpha)\)
- cons = \(\lambda x:\text{int}. \lambda y:(\mu \alpha. \text{unit } + (\text{int } * \alpha)). B((x, y))\)
  - Has type: \(\text{int } \to (\mu \alpha. \text{unit } + (\text{int } * \alpha)) \to (\mu \alpha. \text{unit } + (\text{int } * \alpha))\)
- head = \(\lambda x:(\mu \alpha. \text{unit } + (\text{int } * \alpha)). \text{match } x \text{ with } A.. A(()) \mid B_y. B(y.1)\)
  - Has type: \((\mu \alpha. \text{unit } + (\text{int } * \alpha)) \to (\text{unit } + \text{int})\)
- tail = \(\lambda x:(\mu \alpha. \text{unit } + (\text{int } * \alpha)). \text{match } x \text{ with } A.. A(()) \mid B_y. B(y.2)\)
  - Has type: \((\mu \alpha. \text{unit } + (\text{int } * \alpha)) \to (\text{unit } + \mu \alpha. \text{unit } + (\text{int } * \alpha))\)

But our typing rules allow none of this (yet)

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

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

Examples where type variables appear multiple times:

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

Using \(\mu\) types (continued)

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

\[
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2 \\
\hline
\Delta; \Gamma \vdash A(e) : \tau_1 + \tau_2
\]

So we could show

\[
\Delta; \Gamma \vdash A(()) : \text{unit } + (\text{int } * (\mu \alpha. \text{unit } + (\text{int } * \alpha)))
\]

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

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:

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

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)

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:

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

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

\[
\text{unroll} (\text{roll}_{\mu\alpha.\tau} v) \rightarrow v
\]

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

ML datatypes revealed

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 implicit roll

\>
So \text{Foo } e \text{ is really } \text{roll, Foo}(e)
\>
That is, \text{Foo } e \text{ has type } t (\text{the rolled type})

A pattern-match has an implicit unroll

\>
So match \text{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

Metatheory

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

\>
Erasure (no run-time effect): unchanged
\>
Termination: changed!
\>
\((\lambda x:\mu\alpha.\alpha \rightarrow \alpha. x \ x)(\lambda x:\mu\alpha.\alpha \rightarrow \alpha. x \ x)\)
\>
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, 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”