Recursive Types

We could add list types (list(τ)) and primitives ([], ::, match), but we want user-defined recursive types.

Intuition:

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

Which is roughly:

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

- 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 \( \mu \alpha. \tau \)

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

But our typing rules allow none of this (yet)

Where are we

- System F gave us type abstraction
  - code reuse
  - strong abstractions
- 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 \( \mu \alpha. \text{unit} \rightarrow (\text{int} * \alpha) \)
- int list list: \( \mu \alpha. \text{unit} + ((\mu \beta. \text{unit} + (\text{int} * \beta)) * \alpha) \)

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:

\[
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2
\]

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

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 “unfolding” or “unfolding” (equi-recursive).
Return of subtyping

\[ \Gamma \vdash e : \tau' \quad \text{and these subtyping rules:} \]

\[ \frac{\tau \leq \tau'}{\Gamma \vdash e : \tau} \]

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.

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 \( \tau = \text{Foo of int} \mid \text{Bar of int} \times \tau \)

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