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

But our typing rules allow none of this (yet)

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

Examples where type variables appear multiple times:

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

Using \( \mu \) types (continued)

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

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

So we could show

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

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

But we want \( \mu \alpha. \text{unit} + (\text{int} \times \alpha) \)

Notice: \( \text{unit} + (\text{int} \times (\mu \alpha. \text{unit} + (\text{int} \times \alpha))) \) is

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

The key: Subsumption — recursive types are equal to their “unrolling”
Return of subtyping

Can use subsumption and these subtyping rules:

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

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)

Metatheory

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

- Erasure (no run-time effect): unchanged
- Termination: changed!
  - \((\lambda x: \mu \alpha. \alpha \to \alpha. x x)(\lambda x: \mu \alpha. \alpha \to \alpha. x x)\)
  - 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

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:

\[
\begin{align*}
\tau & ::= \ldots | \mu \alpha. \tau \\
e & ::= \ldots | \text{roll}_{\mu \alpha. \tau} e | \text{unroll} e \\
v & ::= \ldots | \text{roll}_{\mu \alpha. \tau} v \\
e & \rightarrow e' & \text{unroll} e & \rightarrow \text{unroll} e' \\
\text{roll}_{\mu \alpha. \tau} e & \rightarrow \text{roll}_{\mu \alpha. \tau} e' & \text{unroll} e & \rightarrow \text{unroll} e' \\
\text{unroll} (\text{roll}_{\mu \alpha. \tau} v) & \rightarrow v \\
\Delta; \Gamma \vdash e : \tau((\mu \alpha. \tau) / \alpha) & \quad \Delta; \Gamma \vdash e : \mu \alpha. \tau \\
\Delta; \Gamma \vdash \text{roll}_{\mu \alpha. \tau} e \equiv \mu \alpha. \tau & \quad \Delta; \Gamma \vdash \text{unroll} e : \tau((\mu \alpha. \tau) / \alpha)
\end{align*}
\]

ML datatypes revealed

How is \(\mu \alpha. \tau\) related to

\[
type t = \text{Foo of int | Bar of int * t}
\]

Constructor use is a “sum-injection” followed by an implicit roll

- So Foo e is really roll, Foo(e)
- That is, Foo e has type t (the rolled type)

A pattern-match has an implicit unroll

- So match e with... is really match unroll e with...

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