Using µ 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:
  \[\text{int} \to ((\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \to (\mu\alpha.\text{unit} + (\text{int} \times \alpha)))\]
- head =
  \(\lambda x:(\mu\alpha.\text{unit} + (\text{int} \times \alpha)). \text{match} x \text{ with } A_\alpha. A() \mid B y. B(y, 1)\)
  Has type: \((\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \text{int})\)
- tail =
  \(\lambda x:(\mu\alpha.\text{unit} + (\text{int} \times \alpha)). \text{match} x \text{ with } A_\alpha. A() \mid B y. B(y, 2)\)
  Has type:
  \((\mu\alpha.\text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + (\mu\alpha.\text{unit} + (\text{int} \times \alpha)))\)

But our typing rules allow none of this (yet)

Using µ 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

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

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

**SUBSUMPTION**

Can use $\Gamma \vdash e : \tau'$ and these subtyping rules:

**FOLD**

$\tau[(\mu\alpha.\tau)/\alpha] \leq \mu\alpha.\tau$

**UNFOLD**

$\mu\alpha.\tau \leq \tau[(\mu\alpha.\tau)/\alpha]$

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

$\mu\alpha.\tau$

$\tau[(\mu\alpha.\tau)/\alpha]$

$\text{fold}[\mu\alpha.\tau]$

$\text{unfold}[\mu\alpha.\tau]$

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.

---

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*.

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:

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

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

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

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