Programming Language Theory

Recursive Types and Existential Types
Looking back, looking forward

Have defined System F.

- Metatheory (what properties does it have)
- What (else) is it good for
- How/why ML is more restrictive and implicit

Next:

- Recursive types (also use type variables, but differently)
- Existential types (dual to universal types)
System F

\[ e ::= c \mid x \mid \lambda x: \tau. e \mid e \ e \mid \Lambda \alpha. \ e \mid e [\tau] \]

\[ \tau ::= \text{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha. \tau \]

\[ v ::= c \mid \lambda x: \tau. \ e \mid \Lambda \alpha. \ e \]

\[ \Gamma ::= \cdot \mid \Gamma, x: \tau \]

\[ \Delta ::= \cdot \mid \Delta, \alpha \]

\[ e \to_{cbv} e' \]

\[ \frac{e_f \to_{cbv} e_f'}{e_f \ e_a \to_{cbv} e_f' \ e_a} \quad \frac{e_a \to_{cbv} e_a'}{v_f \ e_a \to_{cbv} v_f \ e_a'} \quad \frac{(\lambda x: \tau. \ e_b) \ v_a \to_{cbv} e_b[v_a/x]}{(\Lambda \alpha. \ e_b)[\tau_a] \to_{cbv} e_b[\tau_a/\alpha]} \]

\[ \Delta \vdash \tau \]

\[ \frac{\Delta \vdash \tau_a \quad \Delta \vdash \tau_r}{\Delta \vdash \tau_a \to \tau_r} \quad \frac{\alpha \in \Delta}{\Delta \vdash \alpha} \quad \frac{\Delta, \alpha \vdash \tau_r}{\Delta \vdash \forall \alpha. \tau_r} \]

\[ \Delta; \Gamma \vdash e : \tau \]

\[ \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau} \]

\[ \frac{\Delta \vdash \tau_a \quad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \lambda x: \tau_a. \ e_b : \tau_a \to \tau_r} \]

\[ \frac{\Delta; \Gamma \vdash \tau_a \quad \Delta; \Gamma \vdash e : \tau_a \quad \Delta; \Gamma \vdash e_a : \tau_r}{\Delta; \Gamma \vdash e_f \ e_a : \tau_r} \]

\[ \frac{\Delta; \Gamma \vdash \tau_a \quad \Delta; \Gamma \vdash e : \tau \quad \Delta; \Gamma \vdash \forall \alpha. \tau_r \quad \Delta \vdash \tau_a}{\Delta; \Gamma \vdash e_f [\tau_a] : \tau_r[\tau_a/\alpha]} \]
Type abstraction, recursive types, and existential types

- Universal types (a.k.a., System F)
  - code reuse,
  - strong abstractions
  - different from real languages (like ML), but the right model
- Recursive types (different use of type variables)
  - For building unbounded data structures
  - Turing-completeness without a \texttt{fix} primitive
- Existential types (dual to universal types)
  - First-class abstract types
  - Closely related to closures and objects

All this plus type constructors to understand the list-library example.
Goal

Understand what this interface means and why it matters:

```plaintext
type 'a list
val empty : 'a list
val cons : 'a -> 'a list -> 'a list
val unlist : 'a list -> ('a * 'a list) option
val size : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
```

From two perspectives:

1. Library: Write code to implement this partial specification
2. Client: Use code written to implement this partial specification
Recursive Types

We could directly add a list type \((\text{list } \tau)\) and primitives \((\text{nil, } e :: e, \text{unlist } e)\):

\[
\begin{align*}
\text{Gamma Rule:} & \quad \Gamma \vdash e_h : \tau \\
\text{Beta Rule:} & \quad \Gamma \vdash e_t : \text{list } \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{Gamma Rule:} & \quad \Gamma \vdash \text{nil} : \text{list } \tau \\
\text{Beta Rule:} & \quad \Gamma \vdash \text{unlist } e_l : \text{unit } + (\tau \ast \text{list } \tau) \\
\end{align*}
\]
Recursive Types

Can’t add types/primitives for every (possible) recursive data structure; want user defined recursive types.

Intuition:

```
datatype intlist = Empty | Cons of int * intlist
```

Roughly equivalent to:

```
intlist = unit + int * intlist
```

▶ A name for a recursive type seems unavoidable.
▶ But that’s what we said about recursive functions and we used `fix`
▶ Analogously to `fix (\x. e)`, we will introduce `\alpha. \tau`
▶ Each `\alpha` “stands for” the entire `\alpha. \tau`
Mighty $\mu$

$\mu\alpha. \tau$: in $\tau$, the type variable $\alpha$ (bound by $\mu$) stands for $\mu\alpha. \tau$.

Examples (of many possible encodings):
- int list (finite (cbv or cbn) or infinite (cbn)): $\mu\alpha. \text{unit} + (\text{int} \times \alpha)$
- int list (infinite “stream” (cbn)): $\mu\alpha. \text{int} \times \alpha$
- int list (infinite “stream” (cbv)): $\mu\alpha. (\text{unit} \to \text{int} \times \alpha)$
- int list list: $\mu\alpha. \text{unit} + ((\mu\beta. \text{unit} + (\text{int} \times \beta)) \times \alpha)$
- polymorphic list: $\forall \beta. \mu\alpha. \text{unit} + (\beta \times \alpha)$

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

How do we build and use int lists?

Would like:

$\text{empty} = L(\text{)}$.

Has type: $\mu\alpha. \text{unit} + (\text{int} \ast \alpha)$.

$\text{cons} = \lambda h: \text{int}. \lambda t:\ (\mu\alpha. \text{unit} + (\text{int} \ast \alpha))$. $R((h, t))$.

Has type: $\text{int} \to (\mu\alpha. \text{unit} + (\text{int} \ast \alpha)) \to (\mu\alpha. \text{unit} + (\text{int} \ast \alpha))$.

$\text{hd} = \lambda l: (\mu\alpha. \text{unit} + (\text{int} \ast \alpha))$. case $l$ of $L(x)$ => $L(\text{)}$ | $R(y)$ => $R(y.1)$.

Has type: $(\mu\alpha. \text{unit} + (\text{int} \ast \alpha)) \to (\text{unit} + \text{int})$.

$\text{tl} = \lambda l: (\mu\alpha. \text{unit} + (\text{int} \ast \alpha))$. case $l$ of $L(x)$ => $L(\text{)}$ | $R(y)$ => $R(y.2)$.

Has type: $(\mu\alpha. \text{unit} + (\text{int} \ast \alpha)) \to (\text{unit} + (\mu\alpha. \text{unit} + (\text{int} \ast \alpha)))$.
Using $\mu$ types

How do we build and use int lists?

Would like:

- $\text{empty} = \text{L}()$.
  
  Has type: $\mu\alpha. \text{unit} + (\text{int} \times \alpha)$

But our typing rules allow none of this (yet)
Using $\mu$ types

How do we build and use int lists?

Would like:

- empty = $L(\text{()})$.
  Has type: $\mu\alpha. \text{unit} + (\text{int} \times \alpha)$

- cons = $\lambda h: \text{int}. \lambda t:(\mu\alpha. \text{unit} + (\text{int} \times \alpha)). R((h, t))$.
  Has type: $\text{int} \rightarrow (\mu\alpha. \text{unit} + (\text{int} \times \alpha)) \rightarrow (\mu\alpha. \text{unit} + (\text{int} \times \alpha))$
Using $\mu$ types

How do we build and use int lists?

Would like:

1. **empty** = $L(\text{()}).$
   
   Has type: $\mu\alpha. \text{unit} + (\text{int} \times \alpha)$

2. **cons** = $\lambda h:\text{int}. \lambda t:(\mu\alpha. \text{unit} + (\text{int} \times \alpha)). R((h, t))$
   
   Has type: $\text{int} \to (\mu\alpha. \text{unit} + (\text{int} \times \alpha)) \to (\mu\alpha. \text{unit} + (\text{int} \times \alpha))$

3. **hd** = $\lambda l:(\mu\alpha. \text{unit} + (\text{int} \times \alpha)). \text{case } l \text{ of } L(x) \Rightarrow L(\text{()}) \mid R(y) \Rightarrow R(y.1)$
   
   Has type: $(\mu\alpha. \text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \text{int})$
Using \( \mu \) types

How do we build and use int lists?

Would like:

- **empty** = \( L((\)) \).
  Has type: \( \mu \alpha. \text{unit} + (\text{int} \times \alpha) \)

- **cons** = \( \lambda h: \text{int}. \lambda t:(\mu \alpha. \text{unit} + (\text{int} \times \alpha)). R((h, t)) \)
  Has type: \( \text{int} \rightarrow (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \rightarrow (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \)

- **hd** = \( \lambda l:(\mu \alpha. \text{unit} + (\text{int} \times \alpha)). \text{case } l \text{ of } L(x) \Rightarrow L((\)) \mid R(y) \Rightarrow R(y.1) \)
  Has type: \( (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \rightarrow (\text{unit} + \text{int}) \)

- **tl** = \( \lambda l:(\mu \alpha. \text{unit} + (\text{int} \times \alpha)). \text{case } l \text{ of } L(x) \Rightarrow L((\)) \mid R(y) \Rightarrow R(y.2) \)
  Has type: \( (\mu \alpha. \text{unit} + (\text{int} \times \alpha)) \rightarrow (\text{unit} + (\mu \alpha. \text{unit} + (\text{int} \times \alpha))) \)
Using $\mu$ types

How do we build and use int lists?

Would like:

- **empty** = $L(\text{
})$.
  Has type: $\mu\alpha. \text{unit} + (\text{int} \times \alpha)$

- **cons** = $\lambda h: \text{int}. \lambda t:(\mu\alpha. \text{unit} + (\text{int} \times \alpha)). R((h, t))$
  Has type: $\text{int} \to (\mu\alpha. \text{unit} + (\text{int} \times \alpha)) \to (\mu\alpha. \text{unit} + (\text{int} \times \alpha))$

- **hd** = $\lambda l:(\mu\alpha. \text{unit} + (\text{int} \times \alpha)). \text{case } l \text{ of } L(x) \to L(\text{
}) \mid R(y) \to R(y.1)$
  Has type: $(\mu\alpha. \text{unit} + (\text{int} \times \alpha)) \to (\text{unit} + \text{int})$

- **tl** = $\lambda l:(\mu\alpha. \text{unit} + (\text{int} \times \alpha)). \text{case } l \text{ of } L(x) \to L(\text{
}) \mid R(y) \to R(y.2)$
  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 $\mu$ types (continued)

For empty $= L(())$, one typing rule applies:

$$\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2$$

$$\Delta; \Gamma \vdash L(e) : \tau_1 + \tau_2$$

We could show: $\Delta; \Gamma \vdash L(()) : \text{unit} + (\text{int} \ast (\mu \alpha. \text{unit} + (\text{int} \ast \alpha)))$

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

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

Using $\mu$ types (continued)

For empty $= \text{L}(())$, one typing rule applies:

$$
\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_2
\overline{
\Delta; \Gamma \vdash \text{L}(e) : \tau_1 + \tau_2
}
$$

We could show: $\Delta; \Gamma \vdash \text{L}(()) : \text{unit} + (\text{int} * (\mu\alpha. \text{unit} + (\text{int} * \alpha)))$

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

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

Note: $\text{unit} + (\text{int} * (\mu\alpha. \text{unit} + (\text{int} * \alpha)))$ is $\text{(unit + (int * \alpha))[(\mu\alpha. \text{unit} + (\text{int} * \alpha))/\alpha]}$.

The key: Subsumption — recursive types are equal to their “unrolling”
μ and Subtyping

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, cons, hd, and tl their desired types:

- Constructors use ROLL; destructors use UNROLL

Notice how little we did:

- one new form of type (μα. τ); two new subtyping rules

(Skipping: Depth subtyping on recursive types is very interesting.)
Metatheory

Minimal additions. . .
But must reconsider how recursive types change STLC and System F:

- Erasure Theorem (no run-time effect): unchanged
- Termination Theorem: 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
    (and can type-check every closed \(\lambda\) term (including \(Y\)))
- Curry-Howard: What does this tell us about a “logic” with \(\mu\)?
- Type Safety: still type 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” — we could explicitly tell the type-checker when to roll and unroll.

“Iso-recursive” types (remove subtyping, add expressions):

$$
e ::= \cdots | \text{roll}_{\mu\alpha. \tau}(e) | \text{unroll}(e)$$
$$v ::= \cdots | \text{roll}_{\mu\alpha. \tau}(v)$$
$$\tau ::= \cdots | \mu\alpha. \tau$$

$$\frac{e_a \rightarrow_{\text{cbv}} e'_a}{\text{roll}_{\mu\alpha. \tau}(e_a) \rightarrow_{\text{cbv}} \text{roll}_{\mu\alpha. \tau}(e'_a)}$$

$$\frac{e_a \rightarrow_{\text{cbv}} e'_a}{\text{unroll}(e_a) \rightarrow_{\text{cbv}} \text{unroll}(e'_a)}$$

$$\frac{}{\text{unroll}(\text{roll}_{\mu\alpha. \tau}(v_a)) \rightarrow_{\text{cbv}} v_a}$$

$$\frac{\Delta; \Gamma \vdash e_a : \tau[(\mu\alpha. \tau)/\alpha]}{\Delta; \Gamma \vdash \text{roll}_{\mu\alpha. \tau}(e_a) : \mu\alpha. \tau}$$

$$\frac{\Delta; \Gamma \vdash e_a : \mu\alpha. \tau}{\Delta; \Gamma \vdash \text{unroll}(e_a) : \tau[(\mu\alpha. \tau)/\alpha]}$$
Syntax-directed $\mu$ types (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”
  - $\text{unroll}(\text{roll}_{\mu\alpha. \tau(v)}) \rightarrow_{\text{cbv}} v$
- Most languages do some of each

Anything is decidable if you require the code producer to give the implementation enough “hints” about the “proof”.

Matthew Fluet
Programming Language Theory
Lecture 16
ML datatypes revealed

How is $\mu \alpha. \tau$ related to datatype $t = A \text{ of int} \mid B \text{ of int} * t$?

Using a constructor is a “sum-injection” followed by an *implicit roll*. So $A \ e$ is really $\text{roll}_t(A(e))$.
That is, $A \ e$ has type $t$ (the rolled type).

A pattern-match has an *implicit unroll*. So case $e$ of $\cdots$ is really case unroll($e$) of $\cdots$.

This “trick” works because different recursive types use different tags

- type-checker knows *which* type to roll to
μ vs. fix

Can we write a term of type:

$$\forall \alpha. \forall \beta. ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

that behaves like \textbf{fix}?
Can we write a term of type:

$$\forall \alpha. \forall \beta. ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

that behaves like \texttt{fix}?

$$\Lambda \alpha. \Lambda \beta. \lambda f:(\alpha \to \beta) \to (\alpha \to \beta).$$
$$(\lambda x: \mu \zeta. \zeta \to (\alpha \to \beta). f (\lambda y: \alpha. \text{unroll}(x) x y))$$
$$(\text{roll}_{\mu \zeta. \zeta \to (\alpha \to \beta)} (\lambda x: \mu \zeta. \zeta \to (\alpha \to \beta). f (\lambda y: \alpha. \text{unroll}(x) x y)))$$
μ vs. fix

Can we write a term of type:

\[ \forall \alpha. \forall \beta. ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta) \]

that behaves like fix?

- datatype ('a,'b) r = R of ('a,'b) r -> ('a -> 'b);
- datatype ('a,'b) r = R of ('a,'b) r -> 'a -> 'b
- val rollR = fn x => R x;
- val rollR = fn : (('a,'b) r -> 'a -> 'b) -> ('a,'b) r
- val unrollR = fn (R x) => x;
- val unrollR = fn : ('a,'b) r -> ('a,'b) r -> 'a -> 'b
- val fix = fn f =>
  (fn x => f (fn y => (unrollR x) x y))
  (rollR (fn x => f (fn y => (unrollR x) x y)));
- val fix = fn : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
Back to our goal

Understand what this interface means and why it matters:

```ocaml
type 'a list
val empty : 'a list
val cons : 'a -> 'a list -> 'a list
val unlist : 'a list -> ('a * 'a list) option
val size : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
```

Can “implement” this interface, if we expose the definition of list:

```ocaml
empty : ∀α. μβ. unit + (α * β)
cons : ∀α. α → (μβ. unit + (α * β)) → (μβ. unit + (α * β))
unlist : ∀α. (μβ. unit + (α * β)) → unit + (α * (μβ. unit + (α * β)))
size : ∀α. (μβ. unit + (α * β)) → int
...
```

Unsatisfying...
Abstract Types

Define an interface such that:

a well-typed list-client cannot break the list-library abstraction.

- Want to hide the details of how list is implemented.

Why?

- So clients cannot "forge" lists
  - Lists are always created by library
- So clients cannot rely on the concrete implementation details
  - List library can be changed without breaking clients

To simplify the discussion slightly, consider just intlist.

- (list is a type constructor, a function that given a type gives a type.)
The Type-Application Approach

We can hide intlist using type abstraction (like we hid file-handles):

\[(\Lambda \alpha. \lambda x: \tau_1. \text{list}_\text{client}) \ [\tau_2] \text{list}_\text{library}\]

where:

- \(\tau_1\) is \{empty : \alpha; cons : int \to \alpha \to \alpha; unlist : \alpha \to \text{unit} + (\text{int} \times \alpha); \ldots\}\n- \(\tau_2\) is \(\mu \beta. \text{unit} + (\text{int} \times \beta)\)
- \(\text{list}_\text{client}\) projects from record \(x\) to get list functions
- \(\text{list}_\text{library}\) is the record of list functions
Evaluating the Type Application Approach

We can hide \texttt{intlist} using type abstraction:

\[(\Lambda \alpha. \lambda x:\tau_1. \texttt{list\_client})[\tau_2] \texttt{list\_library}\]

Pros:
- Effective
- Straightforward use of System F

Cons:
- The library does not say \texttt{intlist} should be abstract
  - It relies on clients to abstract it
- Different list-libraries have different types ($\tau_2$); can’t choose one at run-time or put them in a data structure:
  - if n>10 then \texttt{hashset\_lib} else \texttt{listset\_lib}
  - Wish: values \textit{produced} by different libraries may have \textit{different} types, but \textit{libraries} can have the same type
The Higher-Order Approach

We can hide `intlist` using type abstraction and work inside-out:

$$
(\forall \gamma. \lambda y: (\forall \alpha. \tau_1 \rightarrow \gamma). y \ [\tau_2] \ list\_library) \\
[\tau_3] (\forall \alpha. \lambda x: \tau_1. \ list\_client)
$$

where:

- $\tau_1$ is still $\{\text{empty} : \alpha; \text{cons} : \text{int} \rightarrow \alpha \rightarrow \alpha; \\
\text{unlist} : \alpha \rightarrow \text{unit} + (\text{int} \times \alpha); \ldots\}$
- $\tau_2$ is still $\mu \beta. \text{unit} + (\text{int} \times \beta)$
- $\tau_3$ is the type of `list\_client`
- `list\_client` still projects from record $x$ to get list functions
- `list\_library` is still the record of list functions

The library takes the client as a parameter ($y$), ensuring it treats $\tau_2$ (the type of lists) as the abstract $\alpha$; reduces in two steps to previous approach!
Evaluating the Higher-Order Approach

We can hide intlist using type abstraction and work inside-out:

\[(\forall \gamma. \lambda y:(\forall \alpha. \tau_1 \rightarrow \gamma). y [\tau_2] \text{list_library}) [\tau_3] (\forall \alpha. \lambda x:\tau_1. \text{list_client})\]

Pros:

- Still in System F
- Can give different list-libraries the same type:
  - \(\forall \gamma. (\forall \alpha. \tau_1 \rightarrow \gamma) \rightarrow \gamma\)

Cons:

- “Structure inversion” (continuation passing)
- Cannot do this with prenex (ML-style) polymorphism.
The OO Approach

Use recursive types and records:

\[
\text{empty} : \mu \alpha. \{ \text{cons} : \text{int} \to \alpha; \\
\quad \text{unlist} : \text{unit} \to (\text{unit} + (\text{int} \times \alpha)); \\
\quad \text{size} : \text{unit} \to \text{int}; \\
\quad \ldots \}\]

empty is an \textit{object}

- a record of functions plus private data

The \texttt{cons} field holds a function that returns a new record of functions.

Implementation uses recursion and “hidden fields” in an essential way

- In ML, free variables are the “hidden fields”
- In OO, private fields or abstract interfaces “hide fields”
Evaluating the OO Approach

Use recursive types and records:

\[
\text{empty : } \mu \alpha. \{ \text{cons : int } \rightarrow \alpha; \quad \text{unlist : unit } \rightarrow (\text{unit} + (\text{int} \ast \alpha)); \quad \text{size : unit } \rightarrow \text{int}; \quad \ldots \}\n\]

Pros:
- It works in popular languages (no explicit type variables)
- List-libraries have the same type

Cons:
- Changed the interface (no big deal?)
- Fails on “strong” binary \((n > 1)\)-ary operations
  - Have to write append in terms of cons and decons
  - Can be \textit{impossible}
The Existential Approach

Achieved our goal two different ways, but each had some drawbacks

There is a direct way to model ADTs that captures their essence quite nicely:
types of the form $\exists \alpha. \tau$.

Formalization on next slide, but we’ll mostly focus on:
▶ the intuition
▶ the idiom (use idea to encode closures (e.g., for callbacks))

Why don’t many real PLs have existential types?
▶ Because other approaches sort-of work?
▶ Because modules work well even if “second-class”?
▶ Because existentials have only been well-understood since mid-1980s and “tech transfer” takes forever?
Existential Types

\[ e ::= \ldots | \text{pack}_{\exists \alpha. \tau}(\tau, e) | \text{unpack } e \text{ as } (\alpha, x) \text{ in } e \]

\[ \nu ::= \ldots | \text{pack}_{\exists \alpha. \tau}(\tau, \nu) \]

\[ \tau ::= \ldots | \exists \alpha. \tau \]

\[
\frac{e_a \rightarrow_{\text{cbv}} e'_a}{\text{unpack } e_a \text{ as } (\alpha, x) \text{ in } e_b \rightarrow_{\text{cbv}} \text{unpack } e'_a \text{ as } (\alpha, x) \text{ in } e_b}
\]

\[
\frac{\text{unpack } \text{pack}_{\exists \alpha. \tau}(\tau_w, \nu_a) \text{ as } (\alpha, x) \text{ in } e_b \rightarrow_{\text{cbv}} e_b[\tau_w/\alpha][\nu_a/x]}{
\frac{\Delta; \Gamma \vdash e_a : \tau[\tau_w/\alpha]}{
\Delta; \Gamma \vdash \text{pack}_{\exists \alpha. \tau}(\tau_w, e_a) : \exists \alpha. \tau
}
}\]

\[
\frac{\Delta; \Gamma \vdash e_a : \exists \alpha. \tau}{\Delta, \alpha; \Gamma, x:\tau \vdash e_b : \tau_r}
\]

\[
\frac{\Delta; \Gamma \vdash \text{unpack } e_a \text{ as } (\alpha, x) \text{ in } e_b : \tau_r}{\Delta; \Gamma \vdash \Delta \vdash \tau_r}
\]
List library with ∃

List library is an existential package:

\[
\text{pack}(\mu\beta. \text{unit} + (\text{int} \times \beta), \text{list\_library})
\]
as \(\exists\alpha. \{\text{empty} : \alpha; \text{cons} : \text{int} \to \alpha \to \alpha; \text{unlist} : \alpha \to \text{unit} + (\text{int} \times \alpha); \ldots\}\)

Another library would “pack” a different type and implementation, but have the same existential type.

Binary operations work fine: add \textbf{append} : \(\alpha \to \alpha \to \alpha\)

Libraries are first-class, but a \textit{use} of a library must be in a scope that “remembers which \(\alpha\)” describes that library.

▷ If two libraries used in same scope, can’t pass the result of one’s \textbf{cons} to the other’s \textbf{unlist} because the two libraries will use \textit{different} type variables.
▷ Good! The two libraries might have \textit{different} implementations.
Closures and Existentials

There is a deep connection between existential types and how closures are used/compiled.

“Call-backs” are the canonical example.

SML:

► Interface:

\[
\text{val onKeyEvent : (int -> unit) -> unit}
\]

► Implementation:

\[
\begin{align*}
\text{val callBacks : (int -> unit) list ref = ref } & \text{ []} \\
\text{val onKeyEvent f = callBacks := f :: (!callBacks)} \\
\text{val keyPress i = List.app (fn f => f i) (!callBacks)}
\end{align*}
\]

Each registered function can have a different \textit{environment} (free variables of different types), yet every function has type \texttt{int->unit}.
Closures and Existentials

C:

- Interface:

```c
typedef struct {
    void* env;
    void (*f)(void*, int);
} * cb_t;

void onKeyEvent(cb_t);
```

- Implementation (assuming a list library):

```c
list_t callBacks = NULL;
void onKeyEvent(cb_t cb) { callBacks = cons(cb, callBacks); }
void keyPress(int i) {
    for (list_t lst = callBacks; lst; lst = lst->tl) {
        lst->hd->f(lst->hd->env, i);
    }
}
```

Standard problems using subtyping ($t^* \leq \text{void}^*$) instead of $\alpha$:

- Client must provide an $f$ that casts back to $t^*$
- Typechecker lets library pass any pointer to $f$
Closures and Existentials

Cyclone: a safe dialect of C

- (has $\forall \alpha. \tau$ and $\exists \alpha. \tau$, but not closures)

- Interface:

  ```c
  typedef struct {
    a env;
    void (*f)(a,int);
  } *cb_t;

  void onKeyEvent(cb_t);
  ```

- Implementation (assuming a list library):

  ```c
  list_t<cb_t> callBacks = NULL;
  void onKeyEvent(cb_t cb) {
    callBacks = cons(cb,callBacks);
  }
  void keyPress(int i) {
    for (list_t<cb_t> lst = callBacks; lst; lst = lst->tl) {
      let {a env, f} = *lst->hd; // pattern-match
      f(env,i); // no other 1st argument to f typechecks!
    }
  }
  ```

Not shown: When creating a cb_t, must prove “the types match up”