Back to our goal

Understand this interface and its nice properties:

type 'a mylist;
val mt_list : 'a mylist
val cons : 'a -> 'a mylist -> 'a mylist
val decons : 'a mylist -> (('a * 'a mylist) option)
val length : 'a mylist -> int
val map : ('a -> 'b) -> 'a mylist -> 'b mylist

So far, we can do it if we expose the definition of

mt_list :
∀ α.μβ. unit + (α * β)
cons: ∀ α.α → (μβ. unit + (α * β)) → (μβ. unit + (α * β))
...

The Type-Application Approach

We can hide myintlist via type abstraction (like we hid file-handles):

(Λα. λx:τ1. list_client) [τ2] list_library

where:
- τ1 is { mt: α,
  cons: int → α → α,
  decons: α → unit + (int * α),
  ...
}
- τ2 is μβ. unit + (int * β)
- list_client projects from record x to get list functions
- list_library is the record of list functions

The OO Approach

Use recursive types and records:

mt_list : μβ. { cons: int → β,
               decons: (unit + (int * β)) → (unit + (int * β)),
               ...
}

mt_list is an object — a record of functions plus private data

The 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"

(See Caml code for a slightly different example)

Abstract Types

Define an interface such that well-typed list-clients cannot break the list-library abstraction
- Hide the concrete definition of type mylist

Why?
- So clients cannot “forge” lists — always created by library
- So clients cannot rely on the concrete implementation, which lets us change the library in ways that we know will not break clients

To simplify the discussion very slightly, consider just myintlist
- mylist is a type constructor, a function that given a type gives a type

Evaluating ADT via Type Application

(Λα. λx:τ1. list_client) [τ2] list_library

Plus:
- Effective
- Straightforward use of System F

Minus:
- The library does not say myintlist should be abstract
- It relies on clients to abstract it
- Can be “fixed” with a “structure inversion” (passing client to the library), but cure arguably worse than disease
- Different list-libraries have different types, so can’t choose one at run-time or put them in a data structure:
  - if n>10 then haabset_lib else listset_lib
  - Wish: values produced by different libraries must have different types, but libraries can have the same type
Evaluating the Closure/OO Approach

Plus:
- It works in popular languages (no explicit type variables)
- Different list-libraries have the same type

Minus:
- 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 impossible
  (silly example: see type \(t.2\) in ML file)

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 \(\exists \alpha. \tau\)

- Next slide has a formalization, but we’ll mostly focus on
  - The intuition
  - How to use the idea to encode closures (e.g., for callbacks)

- Why don’t many real PLs have existential types?
  - Because other approaches kinda work?
  - Because modules work well even if “second-class”?
  - Because have only been well-understood since the mid-1980s and “tech transfer” takes forever and a day?

Existential Types

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

- \(\Delta;\Gamma \vdash e \ to \ e'\)
  
  \[
  \text{pack } \tau_1, e \ as \ \exists \alpha. \tau_2 \rightarrow \text{pack } \tau_1, e' \ as \ \exists \alpha. \tau_2
  \]

  \[
  \text{unpack } e \ as \ \alpha, x \ in \ e_2 \rightarrow \text{unpack } e' \ as \ \alpha, x \ in \ e_2
  \]

  \[
  \text{unpack} (\text{pack } \tau_1, v \ as \ \exists \alpha. \tau_2) \ as \ \alpha, x \ in \ e_2 \rightarrow e_2[\tau_1/\alpha][v/x]
  \]

- \(\Delta;\Gamma \vdash e : \tau'[\alpha]
  \]

- \(\Delta;\Gamma \vdash \text{pack } \tau, e \ as \ \exists \alpha. \tau' : \exists \alpha. \tau'\)

- \(\Delta;\Gamma \vdash e_1 : \exists \alpha. \tau'\)
  \(\Delta, \alpha; \Gamma, x : \tau' \rightarrow e_2 : \tau\)

- \(\Delta \vdash \tau \ 

  \alpha \notin \Delta
  \]

- \(\Delta;\Gamma \vdash \text{unpack } e_1 \ as \ \alpha, x \ in \ e_2 : \tau\)

List library with \(\exists\)

The list library is an existential package:

- \(\text{pack} (\mu \alpha. \text{unit} + (\text{int} \times \alpha)), \text{list_library} \ as \ \exists \beta\).
  
  \[
  \{ \text{empty} : \beta, \text{cons} : \text{int} \rightarrow \beta \rightarrow \beta, \text{decons} : \beta \rightarrow \text{unit} + (\text{int} \times \beta), \ldots \}
  \]

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

- Binary operations work fine, e.g., \(\text{append} : \beta \rightarrow \beta \rightarrow \beta\)

Libraries are first-class, but a use of a library must be in a scope that “remembers which \(\beta\)” describes data from that library

- (If use two libraries in same scope, can’t pass the result of one’s \text{cons} to the other’s \text{decons} because the two libraries will use different type variables)

Closures and Existentials

There’s a deep connection between existential types and how closures are used/compiled

- “Call-backs” are the canonical example

Caml:

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

- Implementation:
  
  \[
  \text{let callBacks} : (\text{int} \rightarrow \text{unit}) \ \text{list ref} = \text{ref} []
  \]

  \[
  \text{let onKeyEvent } f = \text{callBacks} := f : (!\text{callBacks})
  \]

  \[
  \text{let keyPress } i = \text{List.iter} \ (\text{fun } f \rightarrow f \ i) \ !\text{callBacks}
  \]

Each registered function can have a different environment (free variables of different types), yet every function has type \text{int} \rightarrow \text{unit}

Closures and Existentials

C:

- \text{typedef} \ {\text{struct} \ {\text{void} * \text{env}; \text{void} \ (*f)(\text{void} *, \text{int});} \} \ast \text{cb_t};

- Interface: \text{void onKeyEvent(cb_t);}

- Implementation (assuming a list library):
  
  \[
  \text{list_t callB acks} = \text{NULL};
  \]

  \[
  \text{void onKeyEvent(cb_t cb)\{callBacks=cons(cb,callBacks);\}}
  \]

  \[
  \text{void keyPress(int i) \{}
  \]

  \[
  \text{for(list_t lst=callB acks; lst; lst=lst->tl)}
  \]

  \[
  \text{lst->hd->f(lst->hd->env, i);}
  \]

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

- Client must provide an \(f\) that downcasts argument back to \(t\)

- Typechecker lets library pass any void* to \(f\)
Closures and Existentials

Cyclone (aka Dan’s thesis): (has ∀α.τ and ∃α.τ but not closures)
typedef struct {<a> ‘a env; void (*f)('a,int);} * cb_t;

- Interface: void onKeyEvent(cb_t);
- Implementation (assuming a list library):
  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> x, y} = *lst->hd; // pattern-match
      y(x,i); // no other argument to y typechecks!
    }
  }
  Not shown: To create a cb_t, the “the types must match up”

Type-and-effect systems

New topic: An elegant framework to extend type systems to track “things that may happen” (effects) during evaluation

Plain-old type systems have judgments like $\Gamma \vdash e : \tau$ to mean:

- $e$ won’t get stuck
- If $e$ produces a value, that value has type $\tau$

Adding effects reuses the “plumbing” of typing rules to compute something about “how $e$ executes”

- There are many things we may want to conservatively approximate
- Example: What exceptions might get thrown
- All effect systems are very similar, especially treatment of functions
- Example: All values have no effect since their “computation” does nothing

First a type system

(In this example, exceptions raise constant strings $s$)

$$
\begin{align*}
\tau & ::= \text{bool} \mid \tau \to \tau \mid \tau \star \tau \\
e & ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e e \mid (e, e) \mid e.1 \mid e.2 \\
\Gamma \vdash e : \tau & \quad \Gamma \vdash x : \Gamma(x) \\
\Gamma, x : \tau_1 \vdash e : \tau_2 & \quad \Gamma \vdash \text{true} : \text{bool} \\
\Gamma, x : \tau_1 \vdash e : \tau_2 & \quad \Gamma \vdash \text{false} : \text{bool} \\
\Gamma \vdash \lambda x. e : \tau_1 \to \tau_2 & \quad \Gamma \vdash e : \tau_1 \to \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \to \tau_2 & \quad \Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \star \tau_2 & \quad \Gamma \vdash e : \tau_1 \star \tau_2 \\
\Gamma \vdash e_1 : \text{bool} & \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_3 : \tau \\
\Gamma \vdash \text{if} \ e_1 \ e_2 : \tau & \quad \Gamma \vdash \text{try} e_1 : \tau \\
\Gamma \vdash \text{raise} s : \tau & \quad \Gamma \vdash \text{try} e_1 : \tau \\
\end{align*}
$$

Add effects

$$
\begin{align*}
\epsilon & ::= \ldots \text{sets of strings} \ldots \\
\tau & ::= \text{bool} \mid \tau \to \tau \mid \tau \star \tau \\
e & ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e e \mid (e, e) \mid e.1 \mid e.2 \\
\Gamma \vdash e : \tau & \quad \Gamma \vdash x : \Gamma(x) ; \epsilon \\
\Gamma, x : \tau_1 \vdash e : \tau_2 ; \epsilon & \quad \Gamma \vdash \text{try} e_1 : \tau_1 ; \epsilon \\
\Gamma \vdash \text{false} : \text{bool} ; \epsilon & \quad \Gamma \vdash \text{false} : \text{bool} ; \epsilon \\
\Gamma \vdash \lambda x. e : \tau_1 \to \tau_2 ; \epsilon & \quad \Gamma \vdash e_1 : \tau_1 ; \epsilon \\
\Gamma \vdash e_2 : \tau_1 ; \epsilon & \quad \Gamma \vdash e_1 : \tau_1 ; \epsilon \\
\Gamma \vdash e : \tau_1 \star \tau_2 ; \epsilon & \quad \Gamma \vdash e_1 : \tau_1 \star \tau_2 ; \epsilon \\
\Gamma \vdash (e_1, e_2) : \tau_1 \star \tau_2 ; \epsilon & \quad \Gamma \vdash e_1 : \tau_1 \star \tau_2 ; \epsilon \\
\Gamma \vdash e_2 : \tau_1 \star \tau_2 ; \epsilon & \quad \Gamma \vdash e_1 : \tau_1 \star \tau_2 ; \epsilon \\
\Gamma \vdash e : \tau_1 \to \tau_2 ; \epsilon & \quad \Gamma \vdash e_1 : \tau_1 \to \tau_2 ; \epsilon \\
\end{align*}
$$

Key facts

Soundness: If $\cdot \vdash e : \tau; e$ and $e$ raises uncaught exception $s$, then $s \in e$

- Corollary to Preservation and Progress (once you define the operational semantics for exceptions)

All effect systems work this way:

- Values effectless
- Functions have latent effects
- Conservative due to if and try/handle

Only a couple rules special to this effect system

- Also, not always sets and $\cup$

More general rules

Every effect system also substantially more expressible via appropriate subsumption:

- Typing rule for subjecting (also useful for Preservation)
- Subtyping of function types is covariant in latent effects

$$
\begin{align*}
\Gamma \vdash e : \tau \quad e \subseteq \epsilon' & \quad \frac{\tau_3 \subseteq \tau_4}{\frac{\tau_1 \to \tau_2 \subseteq \tau_3}{\frac{\epsilon' \to \tau_4}{\tau_1 \to \epsilon' \subseteq \epsilon'}}}
\end{align*}
$$

Not shown: Also want effect polymorphism (type variables ranging over effects) for higher-order functions like map
Other examples

- Definitely terminates (true) or possibly diverges (false)
- Give \( \text{fix } e \) effect \( \text{false} \)
- Give values effect \( \text{true} \)
- Treat \( \cup \) as \( \text{and} \)
- No change to rules for functions, pairs, conditionals, etc.
- What type casts might occur (*)
- Are the right variables used in transactions (*)
- Does code obey a locking protocol (*)
- Does code only access memory regions that haven’t been deallocated (*)
- ...

Really a general way to lift static analysis to higher-order functions

(*) The core technique in a research paper Dan has written, though the idea of using effect systems for this sort of thing is not his
- Key is recognizing “from a mile away” when an effect system is the right tool