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

The Type-Application Approach

We can hide `myintlist` via 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

\[
\begin{align*}
\text{mt} : &\; \alpha, \\
\text{cons} : &\; \text{int} \to \alpha \to \alpha, \\
\text{decons} : &\; \alpha \to \text{unit} + (\text{int} \ast \alpha), \\
\cdots 
\end{align*}
\]

▶ \( \tau_2 \) is \( \mu \beta. \text{unit} + (\text{int} \ast \beta) \)

▶ `list_client` projects from record \( x \) to get list functions
▶ `list_library` is the record of list functions
Evaluating ADT via Type Application

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

Plus:
- Effective
- Straightforward use of System F

Minus:
- The library does not say \text{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 hashset_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 \text{t2} in ML file)

The OO Approach

Use recursive types and records:

\[
\text{mt_list} : \mu\beta. \{ \text{cons} : \text{int} \rightarrow \beta, \\
\text{decons} : \text{unit} \rightarrow (\text{unit} + (\text{int} \times \beta)), \\
... \}
\]

\text{mt_list} is an object — a record of functions plus private data

The \text{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)

The Existential Approach

Achieved our goal two different ways, but each had 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

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

\[
v ::= \ldots | \text{pack } \tau, v \text{ as } \exists \alpha.\tau
\]

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

\[
e \rightarrow e'
\]

\[
\text{pack } \tau_1, e \text{ as } \exists \alpha.\tau_2 \rightarrow \text{pack } \tau_1, e' \text{ as } \exists \alpha.\tau_2
\]

\[
e \rightarrow e'
\]

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

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

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

\[
\Delta; \Gamma \vdash \text{pack } \tau, e \text{ as } \exists \alpha.\tau' : \exists \alpha.\tau'
\]

\[
\Delta; \Gamma \vdash e_1 : \exists \alpha.\tau' \quad \Delta, \alpha; \Gamma, x:\tau' \vdash e_2 : \tau \quad \Delta \vdash \tau' \quad \alpha \not\in \Delta
\]

\[
\Delta; \Gamma \vdash \text{unpack } e_1 \text{ as } \alpha, x \text{ in } e_2 : \tau
\]

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:

val onKeyEvent : (int -> unit) -> unit

▷ Implementation:

let callBacks : (int -> unit) list ref = ref []
let onKeyEvent f = callBacks := f::(!callBacks)
let keyPress i = List.iter (fun f -> f i) !callBacks

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

C:

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

▷ Interface: void onKeyEvent(cb_t);

▷ Implementation (assuming a list library):

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 downcasts argument back to \(t^*\)

▷ Typechecker lets library pass any \text{void}^* to \(f\)
Closures and Existentials

A type-safe variant of C could have $\exists \alpha. \tau$ and let programmers code up closures:

typedef struct {<'a> 'a env; void (*f)(‘a,int);} * cb_t;

- Interface: 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> 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”