CS-XXX: Graduate Programming Languages

Lecture 18 — Existential Types

Dan Grossman
2012

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* `mylist`

`mt_list :` $\forall\alpha.\mu\beta.\textbf{unit} + (\alpha * \beta)$
`cons:` $\forall\alpha.\alpha \rightarrow (\mu\beta.\textbf{unit} + (\alpha * \beta)) \rightarrow (\mu\beta.\textbf{unit} + (\alpha * \beta))$
...

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):

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

where:

- τ_1 is $\begin{array}{ll} \{ & \mathbf{mt} : \alpha, \\ & \mathbf{cons} : \mathbf{int} \to \alpha \to \alpha, \\ & \mathbf{decons} : \alpha \to \mathbf{unit} + (\mathbf{int} * \alpha), \\ & \ldots \\ \} & \end{array}$
- τ_2 is $\mu\beta.\mathbf{unit} + (\mathbf{int} * \beta)$
- $list\_client$ projects from record x to get list functions
- $list\_library$ is the record of list functions

Evaluating ADT via Type Application

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

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 hashset_lib else listset_lib`
  - Wish: values *produced* by different libraries must have *different* types, but *libraries* can have the *same* type

The OO Approach

Use recursive types and records:

$$\textbf{mt\_list} : \mu\beta.\ \{\ \textbf{cons} : \textbf{int} \rightarrow \beta,\ \textbf{decons} : \textbf{unit} \rightarrow (\textbf{unit} + (\textbf{int} * \beta)),\ \ldots\}$$

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)

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 `t2` in ML file)

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

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \textbf{pack } \tau, e \textbf{ as } \exists\alpha.\tau \mid \textbf{unpack } e \textbf{ as } \alpha, x \textbf{ in } e \\
v & ::= & \ldots \mid \textbf{pack } \tau, v \textbf{ as } \exists\alpha.\tau \\
\tau & ::= & \ldots \mid \exists\alpha.\tau
\end{array}
$$

$$\frac{e \to e'}{\textbf{pack } \tau_1, e \textbf{ as } \exists\alpha.\tau_2 \to \textbf{pack } \tau_1, e' \textbf{ as } \exists\alpha.\tau_2}$$

$$\frac{e \to e'}{\textbf{unpack } e \textbf{ as } \alpha, x \textbf{ in } e_2 \to \textbf{unpack } e' \textbf{ as } \alpha, x \textbf{ in } e_2}$$

$$\frac{}{\textbf{unpack } (\textbf{pack } \tau_1, v \textbf{ as } \exists\alpha.\tau_2) \textbf{ as } \alpha, x \textbf{ in } e_2 \to e_2[\tau_1/\alpha][v/x]}$$

$$\frac{\Delta; \Gamma \vdash e : \tau'[\tau/\alpha]}{\Delta; \Gamma \vdash \textbf{pack } \tau, e \textbf{ as } \exists\alpha.\tau' : \exists\alpha.\tau'}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \exists\alpha.\tau' \qquad \Delta, \alpha; \Gamma, x{:}\tau' \vdash e_2 : \tau \qquad \Delta \vdash \tau \qquad \alpha \notin \Delta}{\Delta; \Gamma \vdash \textbf{unpack } e_1 \textbf{ as } \alpha, x \textbf{ in } e_2 : \tau}$$

List library with $\exists$

The list library is an existential package:

$$\begin{array}{l} \textbf{pack } (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)), list\_library \textbf{ as} \\ \exists\beta.\ \{\quad \textbf{empty} : \beta, \\ \qquad\quad \textbf{cons} : \textbf{int} \rightarrow \beta \rightarrow \beta, \\ \qquad\quad \textbf{decons} : \beta \rightarrow \textbf{unit} + (\textbf{int} * \beta), \\ \qquad\quad \ldots\} \end{array}$$

Another library would "pack" a *different* type and implementation, but have the *same* overall type

Binary operations work fine, e.g., **append :** $\beta \rightarrow \beta \rightarrow \beta$

Libraries are first-class, but a *use* of a library must be in a scope that "remembers which β" describes data from that library

- (If use two libraries in same scope, can't pass the result of one's **cons** to the other's **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 *environment* (free variables of different types), yet every function has type `int->unit`

Closures and Existentials

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*`$\leq$`void*`) instead of α:

- Client must provide an `f` that downcasts argument back to `t*`
- Typechecker lets library pass any `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):

```
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”