CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2006

Lecture 13

ML, Recursive Types, and Type Abstraction

Where are we

- System F gave us type abstraction (code reuse, strong abstractions)
- Need to discuss erasure and relate to ML (lecture 12, slides 19–24)
- Recursive Types
  - For building unbounded data structures
  - Turing-completeness without a fix primitive
- Existential types
  - First-class abstract types
  - Close relation to closures and objects

Recursive Types

We could add list types (list(τ)) and primitives ([], ::, match), but we want user-defined recursive types.

Intuition:

```
type intlist = Empty | Cons int * intlist
```

Which is roughly:

```
type intlist = unit + (int * intlist)
```

Seems like a named type is unavoidable.

But that's what we thought with let rec and we used fix.

Instead of **fix** $\lambda x.\ e$, we'll do $\mu\alpha.\tau$.

Mighty μ

In τ, type variable α stands for $\mu\alpha.\tau$, bound by μ.

Examples (of many possible encodings):

- int list (finite or infinite): $\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)$
- int list (infinite "stream"): $\mu\alpha.\mathbf{int} * \alpha$
  - Need laziness (thunking) or mutation to build such a thing
- int list list: $\mu\alpha.\mathbf{unit} + ((\mu\beta.\mathbf{unit} + (\mathbf{int} * \beta)) * \alpha)$

Examples where type variables appear multiple times:

- int tree (data at nodes): $\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha * \alpha)$
- int tree (data at leaves): $\mu\alpha.\mathbf{int} + (\alpha * \alpha)$

Using μ types

How do we build and use int lists ($\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)$)?

We would like:

- empty list = $\mathsf{A}(())$.
  Has type: $\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)$.
- cons = $\lambda x{:}\mathbf{int}.\ \lambda y{:}(\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)).\ \mathsf{B}((x, y))$.
  Has type:
  $\mathbf{int} \to (\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)) \to (\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha))$
- head =
  $\lambda x{:}(\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)).\ \mathbf{match}\ x\ \mathbf{with}\ \mathsf{A}\_.\ \mathsf{A}(())\ |\ \mathsf{B}y.\ \mathsf{B}(y.1)$.
  Has type: $(\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)) \to (\mathbf{unit} + \mathbf{int})$.

But our typing rules allow none of this (yet).

Using μ types continued

For empty list $= \mathbf{A}(())$, one typing rule applies:

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \qquad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2}$$

So we could show
$\Delta; \Gamma \vdash \mathsf{A}(()) : \textbf{unit} + (\textbf{int} * (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)))$
(since $FTV(\textbf{int} * \mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)) = \emptyset \subset \Delta$).

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

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

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

Return of subtyping

So we could use *subsumption* and these subtyping rules:

$$\textsc{Roll}\quad \frac{}{\tau[(\mu\alpha.\tau)/\alpha] \leq \mu\alpha.\tau} \qquad\qquad \textsc{Unroll}\quad \frac{}{\mu\alpha.\tau \leq \tau[(\mu\alpha.\tau)/\alpha]}$$

Subtyping can "roll" or "unroll" a recursive type. (Depth subtyping on recursive types is very interesting.)

Can now give empty-list, cons, and head the types we want: Constructors use roll, destructors use unroll.

Notice how little we did: One new form of type ($\mu\alpha.\tau$) and two new subtyping rules.

Metatheory

Despite our minimal additions, we must reconsider how recursive types change ST$\boldsymbol{\lambda}$C 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 λ term)
- Safety: still safe, but Canonical Forms harder
- Inference: Shockingly efficient for "ST$\boldsymbol{\lambda}$C plus μ". (A great contribution of PL theory with applications in OO and XML-processing languages.)

Syntax-directed μ types

Recursive types via subsumption “seems magical” – we can also do it explicitly by telling the type-checker how to roll and unroll.

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

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mu\alpha.\tau \\ e & ::= & \ldots \mid \mathbf{roll}_{\mu\alpha.\tau}\ e \mid \mathbf{unroll}\ e \\ v & ::= & \ldots \mid \mathbf{roll}_{\mu\alpha.\tau}\ v \end{array}$$

$$\frac{e \rightarrow e'}{\mathbf{roll}_{\mu\alpha.\tau}\ e \rightarrow \mathbf{roll}_{\mu\alpha\tau}\ e'} \qquad \frac{e \rightarrow e'}{\mathbf{unroll}\ e \rightarrow \mathbf{unroll}\ e'}$$

$$\frac{}{\mathbf{unroll}\ (\mathbf{roll}_{\mu\alpha.\tau}\ v) \rightarrow v}$$

$$\frac{\Delta; \Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha]}{\Delta; \Gamma \vdash \mathbf{roll}_{\mu\alpha.\tau}\ e : \mu\alpha.\tau} \qquad \frac{\Delta; \Gamma \vdash e : \mu\alpha.\tau}{\Delta; \Gamma \vdash \mathbf{unroll}\ e : \tau[(\mu\alpha.\tau)/\alpha]}$$

Syntax-directed, cont'd

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 does $\mu\alpha.\tau$ relate to `type t = Foo of int | Bar of int * t`...

Using a constructor is a "sum-injection" then *implicit roll*.
So `Foo` e is really **roll**$_{\mathtt{t}}$ **Foo**(e).
That is, `C` e has type `t` (the rolled type).

A pattern-match has an *implicit unroll*.
So `match` e `with...` is `match` **unroll** e `with...`

This "trick" works because different recursive types use different tags (so we know what type to roll to).

Back to our goal

We are understanding 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
```

We can now do it, *if we expose the definition of* `mylist`.

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

(Can implement these functions in System F with μ)

Abstract Types

So that clients cannot "forge" lists or rely on their implementation (breaking code if we change the type definition), we want to hide what `mylist` actually is.

Define an interface such that (well-typed) list-clients cannot break the list-library abstraction.

To simplify the discussion (very slightly), we'll consider just `myintlist`.

(`mylist` is a *type constructor*, a function that given a type gives an (abstract-to-the-client) type).

The Type-Application Approach

We can hide `myintlist` like we hid file-handles:

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

where:

- τ_1 is
  $\{\textbf{mt} : \alpha,$
  $\textbf{cons} : \textbf{int} \rightarrow \alpha \rightarrow \alpha,$
  $\textbf{decons} : \alpha \rightarrow \textbf{unit} + (\textbf{int} * \alpha),$
  $\ldots\}$
- τ_2 is $\mu\beta.\textbf{unit} + (\textbf{int} * \beta)$
- $\mathit{list\_client}$ projects from record x to get list functions

Evaluating ADT via Type Application

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

Plus:

- Effective
- Straightforward use of System F

Minus:

- The library does not say `myintlist` should be abstract; it relies on clients to abstract it.
- Cannot put a bunch of list-libraries in a data structure because they have different types.
  - Lists produced by different libraries must have different types, but libraries can have the same type.

The OO Approach

mt_list :

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

mt_list is an *object* (a record of functions plus private state).

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.)

Evaluation Closure/OO Approach

Plus:

- It works in popular languages (no explicit type variables).
- 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

We 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 $\exists\alpha.\tau$.

Can be formalized, but we'll just show the idea and how we can use it to encode closures (e.g., for callbacks).

(Come ask me if you want to see the semantics and typing rules.)

Preaching: Existential types have been around for over 20 years. They are not that complicated. They should be in our PLs.

Our library with $\exists$

$\textbf{pack}\ (\mu\alpha.\textbf{unit} + (\textbf{int} * \alpha)), list\_library\ \textbf{as}$
$\exists\beta.\{\textbf{mt} : \beta,$
$\quad \textbf{cons} : \textbf{int} \rightarrow \beta \rightarrow \beta,$
$\quad \textbf{decons} : \beta \rightarrow \textbf{unit} + (\textbf{int} * \beta), \ldots\}$

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

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

Binary operations work fine: add **append** : $\beta \rightarrow \beta \rightarrow \beta$

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!=NULL; 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 casts back to `t*`.
- Typechecker lets library pass any pointer to `f`.

Closures and Existentials

Cyclone (aka Dan's thesis): (has $\forall\alpha.\tau$ and $\exists\alpha.\tau$ but not closures)

```
typedef struct {<`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!=NULL; lst=lst->tl) {
      let {<`a> x, y} = *lst->hd; // pattern-match
      y(x,i); // no other argument to y typechecks!
   }
}
```

Note shown: When creating a `cb_t`, must prove "the types match up".