

CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 12

ML, Recursive Types, and Type Abstraction
(Type Variables Continued)

Where are we

- System F gave us type abstraction (code reuse, strong abstractions)
- Need to relate to ML (lecture 11, slides 20–24)
- 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
- Next time: Finish this, then revenge of mutation and exceptions

File-Security Example again

It wasn't so clear how threads are checked and run...

- Type-check an untrusted thread e : $\cdot; \cdot \vdash e :$
 $\forall \alpha. \{\mathbf{fopen} : \mathbf{string} \rightarrow \alpha, \mathbf{fread} : \alpha \rightarrow \mathbf{int}\} \rightarrow \mathbf{unit}.$
- Type-check fork: $\cdot; \cdot \vdash \mathbf{fork} :$
 $\forall \alpha. (\{\mathbf{fopen} : \mathbf{string} \rightarrow \alpha, \mathbf{fread} : \alpha \rightarrow \mathbf{int}\} \rightarrow \mathbf{unit}) \rightarrow \mathbf{unit}$
- Implement fork v : “enqueue”
 $(v[\mathbf{int}]\{\mathbf{fopen} = \lambda x:\mathbf{string} (\dots), \mathbf{fread} = \lambda x:\mathbf{int} (\dots)\})$

Type-checker ensures a thread calls fread only with ints the same thread obtained from fopen. No run-time per-read check.

Recursive Types

We could add list types ($\text{list}(\tau)$) and primitives ($[], ::, \text{match}$), 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: $\mu\alpha.\mathbf{unit} + ((\mu\beta.\mathbf{unit} + (\mathbf{int} * \beta)) * \alpha)$

Type variables can appear multiple times (for trees, etc.)

Using μ types

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

We would like:

- empty list = $\text{inl}()$. Has type: $\mu\alpha.\text{unit} + (\text{int} * \alpha)$.
- cons = $\lambda x:\text{int}.\ \lambda y:(\mu\alpha.\text{unit} + (\text{int} * \alpha)).\ \text{inr}((x, y))$.
Has type:
 $\text{int} \rightarrow (\mu\alpha.\text{unit} + (\text{int} * \alpha)) \rightarrow (\mu\alpha.\text{unit} + (\text{int} * \alpha))$
- head (car) =
 $\lambda x:(\mu\alpha.\text{unit} + (\text{int} * \alpha)).\ \text{case } x\ y.\text{inl}() \mid y.\text{inr}(y.1)$.
Has type: $(\mu\alpha.\text{unit} + (\text{int} * \alpha)) \rightarrow (\text{unit} + \text{int})$.

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

Using μ types continued

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

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash_t \tau_2}{\Delta; \Gamma \vdash \mathbf{inl}(e) : \tau_1 + \tau_2}$$

So we could show

$$\Delta; \Gamma \vdash \mathbf{inl}() : \mathbf{unit} + (\mathbf{int} * (\mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)))$$

(since $FTV(\mathbf{int} * \mu\alpha.\mathbf{unit} + (\mathbf{int} * \alpha)) = \emptyset \subset \Delta$).

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

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

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

Return of subtyping

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

ROLL

$$\frac{}{\tau[(\mu\alpha.\tau)/\alpha] \leq \mu\alpha.\tau}$$

UNROLL

$$\frac{}{\mu\alpha.\tau \leq \tau[(\mu\alpha.\tau)/\alpha]}$$

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 λ 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 (can type-check every closed λ term with type $\mu\alpha.\alpha \rightarrow \alpha$)
- Safety: still safe, but Canonical Forms harder
- Inference: Shockingly efficient for “ST λ C plus μ ”. (A great contribution of PL theory with applications in OO and XML-processing languages.)

Where are we

We have defined anonymous, recursive types and used subsumption to make them *equal* to their unrollings.

With products and sums, we were able to encode data structures (e.g., lists) in a natural way.

With universal quantification, that gives us polymorphic container types (like in homework 1).

We don't need to add fix to $ST\lambda C$ any more.

Non-obvious algorithms exist to check subtyping in the presence of μ types quickly ($O(n^2)$) and completely. But ML-like languages don't need them...

Syntax-directed μ types

Recursive types seem much less magical when the expression language tells the type-checker when to roll and unroll.

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

$$\tau ::= \dots \mid \mu\alpha.\tau$$

$$e ::= \dots \mid \mathbf{roll}_{\mu\alpha.\tau} e \mid \mathbf{unroll} e$$

$$v ::= \dots \mid \mathbf{roll}_{\mu\alpha.\tau} v$$

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

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

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

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

$$e \rightarrow e'$$

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

$$\mathbf{unroll} e \rightarrow \mathbf{unroll} e'$$

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

Syntax-directed, cont'd

Type-checking is syntax-directed / No subtyping necessary.

Canonical Forms, Preservation, and Progress are simpler.

Erasure is (a bit) more complicated.

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”.

μ types in practice

Some languages come close to the full power of μ types and subtyping (e.g., Modula-3).

But most are less powerful (just as ML polymorphism is less powerful than System F).

Nonetheless, understanding μ helps you understand language constructs.

For example, O'Camel datatype definitions wrap sum-types and μ -types in together...

```
type t = A | B of int | C of t * int
```

ML datatypes revealed

When you write `A` or `C e`, you are sort of saying **`rollt inA()`** and **`rollt inC(e)`**.

When you write `match e with...` and `e` has type `t`, you are sort of saying `match unroll e with...`

This “trick” is well-defined because different recursive types use different tags (i.e., variant names).

O’Caml has “polymorphic variants” which are much more like our theoretical language.

But there are really clever (fast) implementation techniques when all possible variants are known.

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

(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.

We’ll investigate several ways to do this in System F (and YFPL) before extending System F with *existential types*.

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

The Type-Application Approach

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

$$(\Lambda\beta. \lambda x:\tau_1. \text{list_client}) [\tau_2] \text{list_library}$$

where:

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

Evaluating ADT via Type Application

$(\Lambda\beta. \lambda x:\tau_1. list_client) [\tau_2] list_library$

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 should have different types, but libraries can have the same type.

The Higher-Order Approach

$(\Lambda\gamma. \lambda y:(\forall\beta.\tau_1 \rightarrow \gamma). y [\tau_2] \textit{list_library})$
 $[\tau_3] \Lambda\beta. \lambda x:\tau_1. \textit{list_client}$

where:

- τ_2 is still $\mu\alpha.\textit{unit} + (\textit{int} * \alpha)$
- τ_1 is still
 $\{\textit{mt} : \beta, \textit{cons} : \textit{int} \rightarrow \beta \rightarrow \beta,$
 $\textit{decons} : \beta \rightarrow \textit{unit} + (\textit{int} * \beta), \dots\}$
- τ_3 is the type of *list_client*
- *list_client* still projects functions from record x

The library takes the client as a parameter, ensuring it treats τ_2 as the abstract β , reducing in two steps to previous approach!

Evaluating Higher-Order Approach

$(\Lambda\gamma. \lambda y:(\forall\beta.\tau_1 \rightarrow \gamma). y [\tau_2] \textit{list_library})$
 $[\tau_3] \Lambda\beta. \lambda x:\tau_1. \textit{list_client}$

Plus:

- Still in System F
- Can give different list-libraries the same type.

Minus:

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

In practice, really relies on closures (so clients can have free variables)

The Closure/OO Approach

$\tau_1 =$

$\mu\alpha.\{\text{cons} : \text{int} \rightarrow \alpha, \text{decons} : \text{unit} \rightarrow (\text{unit} + (\text{int} * \alpha)), \dots\}$

mt_list : τ_1

τ_1 describes objects with methods like **cons** and **decons**.

Implementation uses recursion and private state (OO fields or function's free variables) in an essential way.

(See O'Caml code.)

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: lists with only cons, average, and append)

The Existential Approach

We achieved our goal three 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 $\exists\alpha.\tau$.

We have a way to introduce them (pack) and a way to eliminate them (unpack).

The key idea (that clients cannot know the abstract type) comes through in how we type-check unpack...

Existential Types

$e ::= \dots \mid \text{pack } \tau, e \text{ as } \exists\alpha.\tau \mid \text{unpack } e \text{ as } \alpha, x \text{ in } e$

$v ::= \dots \mid \text{pack } \tau, v \text{ as } \exists\alpha.\tau \quad \tau ::= \dots \mid \exists\alpha.\tau$

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

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

$$\frac{\Delta; \Gamma \vdash e_1 : \exists\alpha.\tau' \quad \Delta, \alpha; \Gamma, x:\tau' \vdash e_2 : \tau \quad \Delta \vdash_{\text{t}} \tau}{\Delta; \Gamma \vdash \text{unpack } e_1 \text{ as } \alpha, x \text{ in } e_2 : \tau}$$

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

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

$$e \rightarrow e'$$

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

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

Our library with \exists

pack ($\mu\alpha.\text{unit} + (\text{int} * \alpha)$), *list_library* as
 $\exists\beta.\{\text{mt} : \beta,$
cons : $\text{int} \rightarrow \beta \rightarrow \beta,$
decons : $\beta \rightarrow \text{unit} + (\text{int} * \beta), \dots\}$

Clients access the functions in the body of an `unpack`.

Libraries are first-class.

(If `unpack` two libraries in same scope, can't pass the result of one's `cons` to the other's `decons` because the two `unpacks` will use different type variables.)

\rightarrow and \exists

With the higher-order approach, we essentially encoded \exists with \forall and \rightarrow .

We can also encode \rightarrow with \forall , \exists , and *closed functions* (like in *C*). (May come back to this.)

So in a language with explicit existential types, we don't need the compiler to implement closures for us.

Preaching: Existential types have been known as useful for describing ADTs for over 20 years. They are not that complicated. They should be in our PLs.