CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2009

Lecture 13— More Parametric Polymorphism; Recursive Types; Type Abstraction

Where are we

Have defined System F. Now:

- Metatheory (what properties does it have)
- What (else) is it good for
- How/why ML is more restrictive and implicit

Then:

- Recursive types (also use type variables, but differently)
- Existential types (dual to universal types)

Metatheory

- Type-safe (need a Type Substitution Lemma)
- All programs terminate (shocking! we saw id [τ] id)
- Parametricity, theorems for free
  - Example: If $\cdot;\cdot \vdash e : \forall\alpha.\forall\beta.(\alpha * \beta) \rightarrow (\beta * \alpha)$, then e is equivalent to $\Lambda\alpha.\ \Lambda\beta.\ \lambda x{:}\alpha * \beta.\ (x.2, x.1)$.

    Every term with this type is the swap function!!

    Intuition: e has no way to make an α or a β and it cannot tell what α or β are or raise an exception or diverge...
- Types do not affect run-time behavior

Note: Mutation "breaks everything"

depth subtyping: hw4, termination: hw3, parametricity: hw4

Security from safety?

Example: A process e should not access files it did not open (fopen can check permissions)

Type-check an untrusted thread e:
$\cdot;\cdot \vdash e : \forall\alpha.\{\textbf{fopen : string} \rightarrow \alpha, \textbf{fread : } \alpha \rightarrow \textbf{int}\} \rightarrow \textbf{unit}$.

Type-checker ensures that a process won't "forge a file handle" and pass it to `fread`.

So `fread` doesn't need to check (faster), file handles don't need to be encrypted (safer), etc.

Moral of Example

In STλC, type safety just means not getting stuck.

With type abstraction, it enables secure interfaces!

Suppose we (the system library) implement file-handles as ints. Then we instantiate α with **int**, but untrusted code *cannot tell*.

Memory safety is a necessary but insufficient condition for language-based *enforcement of strong abstractions*.

Has anything changed?

We said polymorphism was about "many types for same term", but for clarity and easy checking, we changed the syntax via $\boldsymbol{\Lambda\alpha.\ e}$ and $\boldsymbol{e\ [\tau]}$ and the operational semantics via type substitution.

Claim: The operational semantics did not "really" change; types need not exist at run-time.

More formally: There is a translation from System F to the untyped lambda-calculus (with constants) that *erases* all types and produces an equivalent program.

Strengthened induction hypothesis: If $\boldsymbol{e \longrightarrow e_1}$ in System F and $\boldsymbol{erase(e) \longrightarrow e_2}$ in untyped lambda-calculus, then $\boldsymbol{e_2 = erase(e_1)}$.

"Erasure and evaluation commute"

Erasure

Erasure is easy to define:

$$
\begin{aligned}
\boldsymbol{erase(c)} &= \boldsymbol{c} \\
\boldsymbol{erase(x)} &= \boldsymbol{x} \\
\boldsymbol{erase(e_1\ e_2)} &= \boldsymbol{erase(e_1)\ erase(e_2)} \\
\boldsymbol{erase(\lambda x{:}\tau.\ e)} &= \boldsymbol{\lambda x.\ erase(e)} \\
\boldsymbol{erase(\Lambda\alpha.\ e)} &= \boldsymbol{\lambda\_.\ erase(e)} \\
\boldsymbol{erase(e\ [\tau])} &= \boldsymbol{erase(e)\ 0}
\end{aligned}
$$

In pure System F, preserving evaluation order isn't crucial, but it is with fix, exceptions, mutation, etc.

Connection to reality

System F has been one of the most important theoretical PL models since the 1970s and inspires languages like ML.

But you have seen ML polymorphism and it looks different. In fact, it is an implicitly typed restriction of System F.

And these two things ((1) implicit, (2) restriction) have everything to do with each other.

Restrictions

- All types have the form $\forall \alpha_1, \ldots, \alpha_n.\tau$ where $n \geq 0$ and τ has no $\forall$. (Prenex-quantification; no first-class polymorphism.)
- Only let (rec) variables (e.g., x in `let x = e1 in e2`) can have polymorphic types. So $n = 0$ for function arguments, pattern variables, etc. (Let-bound polymorphism)
  - So cannot (always) desugar let to λ in ML.
- For `let rec f x = e1 in e2`, the variable `f` can have type $\forall \alpha_1, \ldots, \alpha_n.\tau_1 \rightarrow \tau_2$ only if every use of `f` in `e1` instantiates each α_i with α_i. (No polymorphic recursion)
- Let variables can be polymorphic only if `e1` is a "syntactic value"
  - a variable, constant, function definition, ...
  - Called the "value restriction"

Why?

ML-style polymorphism can seem weird after you have seen System F. And the restrictions do come up in practice, though tolerable.

- Type inference for System F (given untyped e, is there a System F term e' such that $erase(e') = e$) is undecidable (1995).
- Type inference for ML with polymorphic recursion is undecidable (1992).
- Type inference for ML is decidable and efficient in practice, though pathological programs of size $O(n)$ and run-time $O(n)$ can have types of size $O(2^{2^n})$.
- The type inference algorithm (which many of you have seen in AI!) is *unsound* in the presence of ML-style mutation, but the value-restriction restores soundness.

Recovering lost ground?

Extensions to the ML type system to be closer to System F are judged by:

- Soundness: Do programs still not get stuck?
- Conservatism: Does every old ML program still type-check?
- Power: Does it accept all/most programs from System F?
- Convenience: Are many new types still inferred?

Where are we

- System F gave us type abstraction
  - code reuse, strong abstractions
  - different from real languages (like ML), but the right foundation
- Recursive Types
  - For building unbounded data structures
  - Turing-completeness without a fix primitive
- Existential types
  - First-class abstract types
  - Closely related to closures and objects

All this plus type constructors to understand our list-library example

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
- Analogously to **fix** $\lambda x.\ e$, we'll do $\mu\alpha.\tau$
  - Each α "stands for" entire $\mu\alpha.\tau$

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

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

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

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

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

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.

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.

(Skipping: Depth subtyping on recursive types is very interesting.)

Metatheory

Despite our minimal additions, we must reconsider how recursive types change STλC and System F:

- Erasure (no run-time effect): unchanged
- Termination: changed!
  - $(\lambda x{:}\mu\alpha.\alpha \to \alpha.\ x\ x)(\lambda x{:}\mu\alpha.\alpha \to \alpha.\ x\ x)$
  - 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 "STλC 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):

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

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

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

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

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 is $\mu\alpha.\tau$ related 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 $\textbf{roll}_{\texttt{t}}$ $\textbf{Foo}(e)$.
That is, `Foo` e has type `t` (the rolled type).

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

This "trick" works because different recursive types use different tags – so we know *which* 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` : $\forall\alpha.\mu\beta.\mathbf{unit} + (\alpha * \beta)$

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

...

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 a 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
  $\{\mathbf{mt} : \alpha,$
  $\mathbf{cons} : \mathbf{int} \to \alpha \to \alpha,$
  $\mathbf{decons} : \alpha \to \mathbf{unit} + (\mathbf{int} * \alpha),$
  $\ldots\}$
- τ_2 is $\mu\beta.\mathbf{unit} + (\mathbf{int} * \beta)$
- $\mathit{list\_client}$ 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
- 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`
  - Would prefer: values *produced* by different libraries must have *different* types, but *libraries* can have the *same* type

The OO Approach

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

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.)

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)

Our library with $\exists$

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

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

Binary operations work fine: add **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 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 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> '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: When creating a `cb_t`, must prove "the types match up"