CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 11

Universally Quantified Types (Parametric Polymorphism)

Where are we

- Lambda-calculus let us model functions and scope
- Types let us avoid getting stuck without encoding ints, records, etc.
- Needed fix just to be Turing-Complete, still had to duplicate a lot of code
- Subtyping allowed some code reuse
  - primitive notions (e.g., wider records)
  - lifted to other types (e.g., functions, deeper records)
- Today: Types of the form $\forall \alpha.\tau$
  - uses, theory, connection to ML
- Next Time: Recursive data structures (beyond λ and fix)

The Goal

```
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
```

Amazing facts about this parametric, abstract interface:

1. Code is reusable (e.g., `int`, `int*int`, polymorphic `double_cons`)

The Goal

```
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
```

Amazing facts about this parametric, abstract interface:

2. Clients can't break safety or fail a downcast (cf. Java Vectors) (type system prevents heterogeneous lists)

The Goal

```
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
```

Amazing facts about this parametric, abstract interface:

3. No client can break the abstraction (can reimplement mylists differently, cf. "pure" Scheme or C++)

The Goal

```
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
```

Amazing facts about this parametric, abstract interface:

4. The implementation must be "the same" for any "type instantiation" (i.e., list-library is "parametric" in its type parameters)

(Try to disprove this. Proof is like our normalization proof.)

The plan

Will build an "anonymous types" version of this library over the next couple lectures, while doing much more.

- Relate the (more powerful but cumbersome) formal language to ML

Key technique: type variables (α, β, ...)

- universal for functions: $\forall\alpha.\forall\beta.(\alpha \rightarrow \beta)\ldots$.
- existential for ADTs (cf. closures, objects): $\exists\alpha.(\alpha * (\alpha \rightarrow \textbf{int}))$.
- recursive for data-structures (cf. τ `list`): $\mu\alpha.(\texttt{unit} + (\tau * \alpha))$

But one step at a time...

Simpler examples

id $= \boldsymbol{\lambda x : \tau_1 .\ x}$

apply $= \boldsymbol{\lambda x : \tau_1 .\ \lambda f : \tau_1 \rightarrow \tau_2 .\ f\ x}$

twice $= \boldsymbol{\lambda x : \tau_1 .\ \lambda f : \tau_1 \rightarrow \tau_1 .\ f\ (f\ x)}$

In ST$\boldsymbol{\lambda}$C we had to *choose* a particular $\boldsymbol{\tau_1}$ and $\boldsymbol{\tau_2}$.

In ML, we can use type variables, but what does $\boldsymbol{\lambda x : \alpha .\ x}$ mean?? (Want $\boldsymbol{\forall \alpha . (\alpha \rightarrow \alpha)}$)

Just as ST$\boldsymbol{\lambda}$C was concered with the scope of term variables, our new system will be concerned with the scope of type variables...

Typing Types (sort of)

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \forall\alpha.\tau \mid \textbf{int} \mid \tau \rightarrow \tau \mid \tau * \tau \mid \ldots \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\ \Delta & ::= & \cdot \mid \Delta, \alpha \end{array}$$

Consider types with free type variables meaningless (what is α?)

$$\frac{\alpha \in \Delta}{\Delta \vdash_t \alpha} \qquad \frac{}{\Delta \vdash_t \textbf{int}} \qquad \frac{\Delta \vdash_t \tau_1 \quad \Delta \vdash_t \tau_2}{\Delta \vdash_t \tau_1 \rightarrow \tau_2} \qquad \frac{\Delta, \alpha \vdash_t \tau}{\Delta \vdash_t \forall\alpha.\tau}$$

$\forall\alpha.\tau$ is α-convertible and we define $\tau[\tau'/\alpha]$ just like $e[e'/x]$

We want $\Delta;\Gamma \vdash e : \tau$ to imply $\Delta \vdash_t \tau$ (and $\Delta \vdash_t \tau'$ for all τ' such that $\Gamma(x) = \tau'$)

But then $\lambda x : \alpha.\ x$ won't type-check!

Typing Terms

Type-checking now looks like $\Delta; \Gamma \vdash e : \tau$; whole programs should have no free variables or type variables: $\cdot; \cdot \vdash e : \tau$

Function rule:

$$\frac{\Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash_{\mathsf{t}} \tau_1}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2}$$

How to introduce type variables? Explicit type abstraction:

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau_1}{\Delta; \Gamma \vdash \Lambda \alpha.\ e : \forall \alpha.\tau_1}$$

Type Application

How to eliminate type abstractions? Type application (a.k.a. instantiation):

$$\frac{}{(\boldsymbol{\Lambda\alpha.}\ e)[\tau] \longrightarrow e[\tau/\alpha]} \qquad \frac{\boldsymbol{\Delta;\Gamma} \vdash e : \forall\alpha.\tau_1 \qquad \boldsymbol{\Delta} \vdash_{\mathsf{t}} \tau_2}{\boldsymbol{\Delta;\Gamma} \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}$$

(Define $e[\tau/\alpha]$ as expected.)

Example: let id $= \boldsymbol{\Lambda\alpha.\ \lambda x : \alpha.\ x}$

- id has type $\forall\alpha.\alpha \longrightarrow \alpha$
- id [**int**] has type **int** $\longrightarrow$ **int**
- id [**int** $*$ **int**] has type (**int** $*$ **int**) $\longrightarrow$ (**int** $*$ **int**)
- (id $[\forall\alpha.\alpha \longrightarrow \alpha]$) id has type $\forall\alpha.\alpha \longrightarrow \alpha$

More Examples

Let apply1 $= \boldsymbol{\Lambda\alpha.\ \Lambda\beta.\ \lambda x : \alpha.\ \lambda f : \alpha \to \beta.\ f\ x}$

- apply1 has type $\forall\alpha.\forall\beta.\alpha \to (\alpha \to \beta) \to \beta$
- $\cdot; x$:**int** $\to$ **int** $\vdash$ (apply1 [**int**][**int**] **3** x) : **int**

Let apply2 $= \boldsymbol{\Lambda\alpha.\ \lambda x : \alpha.\ \Lambda\beta.\ \lambda f : \alpha \to \beta.\ f\ x}$

- apply2 has type $\forall\alpha.\alpha \to (\forall\beta.(\alpha \to \beta) \to \beta)$
  (impossible in ML)
- $\cdot; x$:**int** $\to$ **string**, y:**int** $\to$ **int** $\vdash$
  ($\underline{\textbf{let}}\ z =$ apply2 [**int**] $\underline{\textbf{in}}\ z\ (z\ \mathbf{3}$ [**int**] $y)$ [**string**] x) :
  string

twice $= \boldsymbol{\Lambda\alpha.\ \lambda x : \alpha.\ \lambda f : \alpha \to \alpha.\ f\ (f\ x)}$ has type
$\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$

The Whole System (called System F)

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda x:\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e[\tau] \\
\tau & ::= & \mathsf{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha.\tau \\
v & ::= & c \mid \lambda x:\tau.\ e \mid \Lambda\alpha.\ e
\end{array}
$$

$$
\frac{e \rightarrow e'}{\begin{array}{c} e\ e_2 \rightarrow e'\ e_2 \\ v\ e \rightarrow v\ e' \\ e[\tau] \rightarrow e'[\tau] \end{array}}
\qquad
\frac{}{(\lambda x{:}\tau.\ e)v \rightarrow e[v/x]}
\qquad
\frac{}{(\Lambda\alpha.\ e)[\tau] \rightarrow e[\tau/\alpha]}
$$

$$
\frac{}{\Delta;\Gamma \vdash x : \Gamma(x)}
\qquad
\frac{}{\Delta;\Gamma \vdash c : \mathsf{int}}
$$

$$
\frac{\Delta;\Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash_{\mathsf{t}} \tau_1}{\Delta;\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \rightarrow \tau_2}
\qquad
\frac{\Delta,\alpha;\Gamma \vdash e : \tau_1}{\Delta;\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\tau_1}
$$

$$
\frac{\Delta;\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau_1}
\qquad
\frac{\Delta;\Gamma \vdash e : \forall\alpha.\tau_1 \qquad \Delta \vdash_{\mathsf{t}} \tau_2}{\Delta;\Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}
$$

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

Where are we

Understand parametric polymorphism and our ML-like list interface.

- Defined System F, saw "simple" examples
- Mentioned some unbelievable theorems

Now:

- Reconsider list example
- "Security"-Related example
- Discuss erasure
- Relate to ML

Our goal, revisited

```
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 give types to all these values (universally quantify over α and β).

If `'a mylist` was an abbreviation for `unit + 'a`, we could give expressions with these types in System F. But

- values of type `unit +` τ have 0 or 1 τ (not a list)
- exposing type definition lets clients break abstraction

Security from safety?

In STλC, type safety just means not getting stuck.

With type abstraction, it enables secure interfaces!
Example: A thread e should not access files it did not open (fopen can check permissions)

$$\Lambda\alpha.\ \lambda\{\textbf{fopen} : \textbf{string} \to \alpha, \textbf{fread} : \alpha \to \textbf{int}\}.\ e$$

Parametricity ensures any value passed to fread came from this thread calling fopen...

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 $e\ [\tau]$ 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 $e \longrightarrow \boldsymbol{e_1}$ in System F and $\boldsymbol{erase}(e) \longrightarrow \boldsymbol{e_2}$ in untyped $\boldsymbol{\lambda}$, then $\boldsymbol{e_2 = erase(e_1)}$.

"Erasure and evaluation commute"

Erasure

Erasure is easy to define—let's do it together:

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

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 early 70s 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)
- 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, or function definition. (The value-restriction)

Why? (Part 1)

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 (allowinng different instantiation in body of recursive functions) 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 $O(2^{2^n})$.

Why? (Part 2)

- 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.

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?

Proposals are getting mature; will probably happen soon.

That was a lot!

We saw System F and discussed its many amazing properties.

We compared System F to ML-style polymorphism, which should make more sense now.

Next up: Recursive types and existential types (which complete our list example)

Then: Mutation (and how it destroys all our nice properties)