CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2005

Lecture 12

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

Understand what this interface means and why it matters:

```
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
```

From two perspectives:

1. Library: Implement code to this specification
2. Client: Use code written to this specification

What The Client Likes

1. Library is reusable. Can make:
   - Different lists with elements of different types
   - New reusable functions outside of library (e.g., `val twocons : 'a -> 'a -> 'a mylist -> 'a mylist`
2. Easier, faster, more reliable than subtyping (cf. Java 1.4 Vector)
   - No downcast to write, run, maybe-fail
3. Library must "behave the same" *for all* "type instantiations"!!
   - `'a` and `'b` held abstract from library functions
   - E.g., with built-in lists: If `foo` has type `'a list -> int`, `foo [1;2;3]` and `foo [(5,4);(7,2);(9,2)]` are totally equivalent! (Never true with downcasts)
   - In theory, means less (re)-integration testing
   - Proof is beyond this course, but not much.

What the Library Likes

1. Reusability. For same reasons client likes it.
2. Abstraction of `mylist` from clients.
   - Clients must "behave the same" *for all* equivalent implementations, even if "hidden definition" of `'a mylist` changes.
   - Clients typechecked knowing only *there exists* a *type constructor* `mylist`
   - Unlike Java, C++, (pure) Scheme, no way to downcast a `t mylist` to, e.g., a pair.

Start simpler

Our interface has a lot going on:

1. Element types *held abstract* from library.
2. List type (constructor) *held abstract* from client.
3. Reuse of type variables "makes connections" among expressions of abstract types.
4. Lists need some form of recursive type STλC has no unbounded data structures (except maybe functions).

Today just consider (1) and (3)

- First using a formal language with explicit type abstraction.
- Then highlight differences with ML.

Note: Much more interesting than "not getting stuck"

Syntax

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

New:

- Type variables
- Types, terms, and contexts to know "what type variables are in scope" (much like we did for term variables)
- Type-applications to *instantiate* polymorphic expressions

Semantics

Our evaluation judgment (e.g., small-step left-right $e \longrightarrow e'$) still looks the same. Just two new rules (note $\boldsymbol{\Lambda\alpha.\ e}$ a value):

$$\frac{e \longrightarrow e'}{e[\tau] \longrightarrow e'[\tau]} \qquad \frac{}{(\boldsymbol{\Lambda\alpha.\ e})[\tau] \longrightarrow e[\tau/\alpha]}$$

(Plus definition of $e[\tau'/\alpha]$ and $\tau[\tau'/\alpha]$ in the straightforward capture-avoiding way.)

Example (using addition):

$$(\boldsymbol{\Lambda\alpha.\ \Lambda\beta.\ \lambda x : \alpha.\ \lambda f{:}\alpha \to \beta.\ f\ x})\ [\textbf{int}]\ [\textbf{int}]\ \mathbf{3}\ (\boldsymbol{\lambda y : \textbf{int}.\ y + y})$$

Typing

Mostly we just get picky about "no free type variables":

- Typing judgment has the form $\boldsymbol{\Delta; \Gamma \vdash e : \tau}$
  (whole program $\cdot; \cdot \vdash e : \tau$).
- Uses helper judgment $\boldsymbol{\Delta \vdash \tau}$ (i.e., $\boldsymbol{FTV(\tau) \subseteq \Delta}$).

New rules:

$$\frac{\boldsymbol{\Delta, \alpha; \Gamma \vdash e : \tau_1}}{\boldsymbol{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\tau_1}} \qquad \frac{\boldsymbol{\Delta; \Gamma \vdash e : \forall\alpha.\tau_1} \qquad \boldsymbol{\Delta \vdash \tau_2}}{\boldsymbol{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}}$$

(Also modify rule for functions so argument type cannot make up type variables (that is what $\boldsymbol{\Lambda\alpha.\ .e}$ is for).)

Example (using addition):

$$\boldsymbol{(\Lambda\alpha.\ \Lambda\beta.\ \lambda x : \alpha.\ \lambda f{:}\alpha \to \beta.\ f\ x)\ [\text{int}]\ [\text{int}]\ 3\ (\lambda y : \text{int}.\ y + y)}$$

Really picky

Allowing free type variables *will* burn the language designer/implementor. It's boring, but too important to omit.

$\boxed{\Delta \vdash \tau}$

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

Want this technical theorem:
If $\Delta; \Gamma \vdash e : \tau$ and $(\forall x \in \mathbf{Dom}(\Gamma).\ \ \Delta \vdash \Gamma(x))$, then $\Delta \vdash \tau$.

That is, "type-checker doesn't let type variables escape scope".

In practice, you put just enough hypotheses of the form $\Delta \vdash \tau$ in your typing rules that the theorem holds.

The Whole Language (called System F)

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

$$\frac{e \to e'}{e\ e_2 \to e'\ e_2} \qquad \frac{e \to e'}{v\ e \to v\ e'} \qquad \frac{e \to e'}{e[\tau] \to e'[\tau]}$$

$$\frac{}{(\lambda x{:}\tau.\ e)v \to e[v/x]} \qquad \frac{}{(\Lambda\alpha.\ e)[\tau] \to e[\tau/\alpha]}$$

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

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

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau_1} \qquad \frac{\Delta;\Gamma \vdash e : \forall\alpha.\tau_1 \quad \Delta \vdash \tau_2}{\Delta;\Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}$$

Examples

Colin's favorite polymorphic function...

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

- id has type $\forall\boldsymbol{\alpha.\alpha \to \alpha}$
- id [**int**] has type **int** $\to$ **int**
- id [**int** $*$ **int**] has type (**int** $*$ **int**) $\to$ (**int** $*$ **int**)
- (id $[\forall\boldsymbol{\alpha.\alpha \to \alpha}]$) id has type $\forall\boldsymbol{\alpha.\alpha \to \alpha}$

In ML you can't do the last one; in System F you can.

More Examples

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

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

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

- applyNew 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** $\vdash$
  (**let** $z =$ applyNew [**int**] **in** $z\ (z\ \mathbf{3}$ [**int**] $y)$ [**string**] x) : **string**

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

- twice has type $\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$
- Cannot be made more polymorphic.

Metatheory

- Type-safe (need a Type Substitution Lemma)
- All programs terminate (shocking!! we saw id $[\tau]$ 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 $\mathbf{\Lambda\alpha.\ \Lambda\beta.\ \lambda x{:}\alpha * \beta.\ (x.2, x.1)}$.
    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
- Stated some unbelievable theorems

Now:

- "Security" example
- Discuss erasure
- Relate to ML

Security from safety?

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

- Type-check an untrusted thread e:
  $\cdot;\cdot \vdash e : \forall\alpha.\{\textbf{fopen : string} \to \alpha, \textbf{fread : } \alpha \to \textbf{int}\} \to \textbf{unit}$.
- Type-check spawn: $\cdot;\cdot \vdash \textbf{spawn :}$
  $\forall\alpha.(\{\textbf{fopen : string} \to \alpha, \textbf{fread : } \alpha \to \textbf{int}\} \to \textbf{unit}) \to \textbf{unit}$
- Implement spawn v: "enqueue"
  $(v[\textbf{int}]\{\textbf{fopen} = \lambda x\textbf{:string. } (\ldots), \textbf{fread} = \lambda x\textbf{:int. } (\ldots)\})$

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

Moral of Example

In STλC, type safety just means not getting stuck.

With type abstraction, it enables secure interfaces!

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 $\boldsymbol{e\ [\tau]}$ 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 $\boldsymbol{\lambda}$, 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 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)
  - 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, ... (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 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?

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: Exceptions

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

In other versions of 505: How to do type inference for ML (algorithm almost fits on a slide).