CSE-505: Programming Languages

Lecture 16 — Parametric Polymorphism

Zach Tatlock
2015

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 partial specification
2. Client: Use code written to this partial 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
   - No downcast to write, run, maybe-fail (cf. Java 1.4 `Vector`)

3. Library must "behave the same" *for all* "type instantiations"!
   - `'a` and `'b` held abstract from library
   - E.g., with built-in lists: If `foo` has type `'a list -> int`, then `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++, R5RS Scheme, no way to downcast a `t mylist` to, e.g., a pair

Start simpler

The 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

This lecture considers just (1) and (3)

- First using a formal language with explicit type abstraction
- Then mention differences from ML

Note: Much more interesting than "not getting stuck"

Syntax

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

New things:

- Terms: Type abstraction and type application
- Types: Type variables and universal types
- Type contexts: what type variables are in scope

Informal semantics

1. $\Lambda\alpha.\ e$: A value that, when used, runs e (with some τ for α)
   - To type-check e, know α is *one* type, but not *which* type
2. $e[\tau]$: Evaluate e to some $\Lambda\alpha.\ e'$ and then run e'
   - With τ for α, but the choice of τ is irrelevant at run-time
   - τ used for type-checking and proof of Preservation
3. Types can use type variables α, β, etc., but only if they're *in scope* (just like term variables)
   - Type-checking will be $\Delta; \Gamma \vdash e : \tau$ using Δ to know what type variables are in scope in e
   - In universal type $\forall\alpha.\tau$, can also use α in τ

Operational semantics

Small-step, CBV, left-to-right operational semantics:

- Note: $\boldsymbol{\Lambda\alpha.\ e}$ is a value

$\boxed{e \to e'}$

Old: $\dfrac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$ $\dfrac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$ $\dfrac{}{(\lambda x{:}\tau.\ e)\ v \to e[v/x]}$

New: $\dfrac{e \to e'}{e[\tau] \to e'[\tau]}$ $\dfrac{}{(\Lambda\alpha.\ e)[\tau] \to e[\tau/\alpha]}$

Plus now have 3 different kinds of substitution, all defined in straightforward capture-avoiding way:

- $e_1[e_2/x]$ (old)
- $e[\tau'/\alpha]$ (new)
- $\tau[\tau'/\alpha]$ (new)

Example

Example (using addition):

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

Example

Example (using addition):

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

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

Example

Example (using addition):

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

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

$$\to\ (\lambda x : \text{int}.\ \lambda f{:}\text{int} \to \text{int}.\ f\ x)\ 3\ (\lambda y : \text{int}.\ y + y)$$

Example

Example (using addition):

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

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

$$\to\ (\lambda x : \textsf{int}.\ \lambda f{:}\textsf{int} \to \textsf{int}.\ f\ x)\ 3\ (\lambda y : \textsf{int}.\ y + y)$$

$$\to\ (\lambda f{:}\textsf{int} \to \textsf{int}.\ f\ 3)\ (\lambda y : \textsf{int}.\ y + y)$$

Example

Example (using addition):

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

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

$$\to\ (\lambda x : \text{int}.\ \lambda f{:}\text{int} \to \text{int}.\ f\ x)\ 3\ (\lambda y : \text{int}.\ y + y)$$

$$\to\ (\lambda f{:}\text{int} \to \text{int}.\ f\ 3)\ (\lambda y : \text{int}.\ y + y)$$

$$\to\ (\lambda y : \text{int}.\ y + y)\ 3$$

Example

Example (using addition):

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

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

$$\to\ (\lambda x : \text{int}.\ \lambda f{:}\text{int} \to \text{int}.\ f\ x)\ 3\ (\lambda y : \text{int}.\ y + y)$$

$$\to\ (\lambda f{:}\text{int} \to \text{int}.\ f\ 3)\ (\lambda y : \text{int}.\ y + y)$$

$$\to\ (\lambda y : \text{int}.\ y + y)\ 3$$

$$\to\ 3 + 3$$

Example

Example (using addition):

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

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

$$\to\ (\lambda x : \text{int}.\ \lambda f{:}\text{int} \to \text{int}.\ f\ x)\ 3\ (\lambda y : \text{int}.\ y + y)$$

$$\to\ (\lambda f{:}\text{int} \to \text{int}.\ f\ 3)\ (\lambda y : \text{int}.\ y + y)$$

$$\to\ (\lambda y : \text{int}.\ y + y)\ 3$$

$$\to\ 3 + 3$$

$$\to\ 6$$

Type System, part 1

Mostly just need to be picky about "no free type variables"

- Typing judgment has the form $\Delta; \Gamma \vdash e : \tau$ (whole program $\cdot; \cdot \vdash e : \tau$)
  - Next slide
- Uses helper judgment $\Delta \vdash \tau$
  - "all *free* type variables in τ are in Δ"

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

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

Rules are boring, but trust me, allowing free type variables is a pernicious source of language/compiler bugs

Type System, part 2

Old (with one technical change to prevent free type variables):

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

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

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau_1}$$

New:

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

Example

Example (using addition):

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

(The typing derivation is rather tall and painful, but just a syntax-directed derivation by instantiating the typing rules)

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

An overly simple polymorphic function...

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

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

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

More Examples

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

- apply1 has type $\forall\alpha.\forall\beta.\alpha \to (\alpha \to \beta) \to \beta$
- $\cdot; g$:**int** $\to$ **int** $\vdash$ (apply1 [**int**][**int**] **3** g) : **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)
- $\cdot; g$:**int** $\to$ **string**, h:**int** $\to$ **int** $\vdash$ (**let** z = apply2 [**int**] **in** z (z **3** [**int**] h) [**string**] g) : **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

What next?

Having defined System F...

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

Metatheory

- Safety: Language is type-safe
  - Need a Type Substitution Lemma
- Termination: All programs terminate
  - Surprising — we saw id [τ] id
- Parametricity, a.k.a. theorems for free
  - Example: If $\cdot;\cdot \vdash e : \forall\alpha.\forall\beta.(\alpha * \beta) \to (\beta * \alpha)$, 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...
- Erasure: Types do not affect run-time behavior

Note: Mutation "breaks everything"

- depth subtyping: hw4, termination: hw3, parametricity: hw5

Security from safety?

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

Require an untrusted process e to type-check as follows:
$\cdot;\cdot \vdash e : \forall\alpha.\{\textbf{fopen : string} \to \alpha, \textbf{fread : } \alpha \to \textbf{int}\} \to \textbf{unit}$

This type 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 simply-typed lambda-calculus, 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*

Are types used at run-time?

We said polymorphism was about "many types for same term", but for clarity and easy checking, we changed:

- The syntax via $\Lambda\alpha.\ e$ and $e\ [\tau]$
- The operational semantics via type substitution
- The type system via Δ

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

More formally: *Erasing* all types from System F produces an equivalent program in the untyped lambda calculus

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

"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.

These two qualifications ((1) implicit, (2) restriction) are deeply related.

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
- In `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" (relaxed partially in OCaml)

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 is *unsound* in the presence of ML-style mutation, but value-restriction restores soundness
  - Based on *unification*

Recovering lost ground?

Extensions to the ML type system to be closer to System F:

- Usually require some type annotations
- Are judged by:
  - Soundness: Do programs still not get stuck?
  - Conservatism: Do all (or most) old ML programs still type-check?
  - Power: Does it accept many more useful programs?
  - Convenience: Are many new types still inferred?