CSE 505: Programming Languages

Lecture 18 — Parametric Polymorphism

Zach Tatlock
Fall 2013

Last Time

Saw structural subtyping

- constraints over record fields
- propagate constraints to "bigger" types
- covariance, contravariance

Provided polymorphism over records with "enough" fields ...
but **at fixed types**.

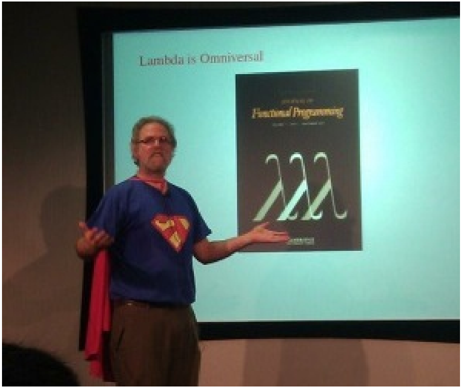
What if code imposes *no constraints* on some types?

This Time: Parametric Polymorphism

Some code just doesn't care what types it's operating over.

You might even say it works *universally*... ???

Before we figure out what that means, a word from a luminary:



MOVIE TIME!


fresh
POP CORN
DELICIOUS
CRISP
fresh
POP CORN
DELICIOUS
NUTRITIOUS
fresh
POP CORN
DELICIOUS
NUTRITIOUS
fresh
POP CORN
DELICIOUS
CRISP

Goal: Everybody Wins!

Understand what this interface means and why it matters:

```
type 'a mylist;
val empty   : '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. Client: Code against this specification
2. Library: Implement this specification

Goal: Client Wins!

1. Reusability (at different types!)
   - 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 `List`)
3. Library must "behave the same" *for all* "type instantiations"!
   - `'a` and `'b` held abstract from library
   - e.g., suppose `foo: 'a list -> int`, then
     `foo [1;2;3]` totally equivalent to `foo [(5,4);(7,2);(9,2)]`
   - Why? Still true if we have downcasts?
   - Proof left as exercise to the reader
   - In theory, means less (re-)integration testing

Goal: Library Wins!

1. Reusability — all the same reasons client likes it
2. Abstraction of `mylist` from clients
   - Clients can only assume interface, no implementation details
   - Free to change/optimize hidden details of `'a mylist`
   - Clients typechecked knowing only:
     there exists some type constructor `mylist`
   - Unlike Java/C++ cannot downcast a `t mylist` to, e.g., a pair

Start Simple

The `mylist` 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 constrains expressions over abstract types
4. Lists need some form of recursive type

We'll focus on (1) and (3):

- First using a formal language with explicit type abstraction
- Then compare and contrast with ML

Note: Much more interesting than "not getting stuck"

Recipe for Extension

1. Add syntax
2. Add semantics
3. Add typing rules
4. Patch up type safety proof

1. Add Syntax

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

Summary of new things:

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

2. Add Semantics

What is this Λ (big lambda) thing? Informally:

1. $\Lambda\alpha.\ e$: a value that takes some τ, plugs it in for α, then runs e
    - type-check e knowing α is *some* type, but not *which* type
2. $e[\tau]$: crunch e down to some $\Lambda\alpha.\ e'$, plug in τ for α, run e'
    - choice of τ is irrelevant at run-time
    - τ used for type-checking and proof of Preservation

What is this $\forall$ (upside down "A") thing? Informally:

Types can use type variables α, β, etc., but only if they're *in scope* (just like term variables)

- Type-checking $\Delta; \Gamma \vdash e : \tau$ uses Δ to scope type vars in e
- universal type $\forall\alpha.\tau$, brings α into scope for τ

2. Add Semantics

Formal, small-step, CBV, left-to-right operational semantics:

- Recall: $\Lambda\alpha.\ e$ is a value

$\boxed{e \to e'}$

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

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

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):

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

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

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

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

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

$$\to\ 3+3$$

$$\to\ 6$$

3. Add Typing Rules

Need to be picky about "no free type variables"

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

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

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

Rules are boring, but smart people found out the hard way that allowing free type variables is a pernicious source of language/compiler bugs.

3. Add Typing Rules

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

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

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

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

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):

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

Ouch.

Just a syntax-directed derivation by instantiating the typing rules.
Still, machines are better suited to this stuff.

System F *(Tah Dah!)*

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

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

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

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

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

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

Examples

Perhaps the simplest polymorphic function...

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

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

In ML you can't do the last one! What?!

More Examples

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

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

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

- apply2 has type $\boldsymbol{\forall\alpha.\alpha \to (\forall\beta.(\alpha \to \beta) \to \beta)}$ (also impossible in ML!)
- $\cdot; g$:**int** $\to$ **string**, h:**int** $\to$ **int** $\vdash$ (<u>**let**</u> $z =$ apply2 [**int**] <u>**in**</u> z (z $\mathbf{3}$ [**int**] h) [**string**] g) : **string**

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

- twice has type $\boldsymbol{\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha}$
- Could this be any more polymorphic?

4. Type Safety and Metatheory

- Safety: System F is type-safe
  - Need a Type Substitution Lemma
- Termination: All programs terminate
  - Even with self application — we saw id $[\tau]$ 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...
  - How many terms have type $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$?

Note: Mutation breaks everything :(

What next?

Now that we have System F...

- What hath we wrought? Example of our mighty new powers.
- How/why ML is more restrictive and implicit.

Security from safety?

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

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

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 STLC, type safety just meant not getting stuck

Type abstraction gives us new powers, e.g. 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:

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

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

Connection to reality... or at least ML

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.

ML 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)

ML Restrictions: 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*

Recover 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?