Programming Language Theory

Parametric Polymorphism
Goal

Understand what this interface means and why it matters:

type 'a list
val empty : 'a list
val cons : 'a -> 'a list -> 'a list
val unlist : 'a list -> ('a * 'a list) option
val size : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list

From two perspectives:
1. Library: Write code to implement this partial specification
2. Client: Use code written to implement this partial specification
What The Client Likes

Library is reusable. Can make:

- Different lists with elements of different types
- New, reusable functions outside of library.
  - `val consTwo: 'a -> 'a -> 'a list -> 'a list`

Easier, faster, and more reliable than subtyping

- No downcast to write, run, and maybe-fail (cf. Java 1.4 Vector)

Library must “behave the same” for all “type instantiations”!

- 'a and 'b held abstract from library functions
- If `size` has type `'a list -> int`
  
  then `size [1,2,3] and size [(1,2),(3,4),(5,6)]` are totally equivalent! (Never true with downcasts)

- In theory, means less (re)-integration testing
- Proof is beyond this course (but very interesting (cf. “Theorems for Free”)).
What the Library Likes

Reusability — For same reasons as client.

Abstraction of list from clients
► Clients must “behave the same” *for all* equivalent implementations, even if “hidden definition” of 'a list changes
► Clients typechecked knowing only *there exists* a type constructor `list`
► Clients cannot “see” or “make assumptions about” definition of list
  ► Unlike some langs, no way to downcast a `t list` 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

For now, 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

\[ e ::= c \mid x \mid \lambda x:\tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e [\tau] \]

\[ v ::= c \mid \lambda x:\tau. \ e \mid \Lambda \alpha. \ e \]

\[ \Gamma ::= \cdot \mid \Gamma, x:\tau \]

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \tau \]

\[ \Delta ::= \cdot \mid \Delta, \alpha \]

New things:

- Type variables: \( \alpha \)
- Terms: type abstraction and type applications
- Types: type variables and universal types
- Type context: to know “what type variables are in scope”
  - similar to handling of term variables with term context

Same “concrete-syntax ambiguities” resolutions.
Informal Semantics

- $\Lambda \alpha. \, e_b$: A value that, when applied, runs $e_b$ (with some type $\tau$ for $\alpha$)
  - To type-check $e_b$, know $\alpha$ is some type, but not which type

- $e_f \left[ \tau_a \right]$: Evaluate $e_f$ to some $\Lambda \alpha. \, e_b$ and then run $e_b$ (with type $\tau_a$ for $\alpha$)
  - But, the choice of $\tau_a$ is irrelevant at run-time
  - $\tau_a$ used for type-checking and proof of Preservation (but not of Progress)

- Types can use type variables $\alpha$, $\beta$, etc., but only ones that are *in scope* (just like term variables)
  - Type-checking judgement will be $\Delta; \Gamma \vdash e : \tau$, using $\Delta$ to know what type variables are in scope for $e$
  - In a universal type $\forall \alpha. \tau$, can also use $\alpha$ in $\tau$

- Work with terms “up to renaming of bound type variables” (“up to alpha-conversion”)
  - In $\Lambda \alpha. \, e$, $\alpha$ is bound in $e$
  - In $\forall \alpha. \tau$, $\alpha$ is bound in $\tau$
Operational Semantics

Small-step, *call-by-value (CBV)*, left-to-right operational semantics:

\[
e \rightarrow_{cbv} e'
\]

\[
(\lambda x: \tau. e_b) v_a \rightarrow_{cbv} e_b[v_a/x]
\]

\[
e_f \rightarrow_{cbv} e'_f
\]

\[
e_f e_a \rightarrow_{cbv} e'_f e_a
\]

\[
v_f e_a \rightarrow_{cbv} v_f e'_a
\]

\[
(\Lambda \alpha. e_b) [\tau_a] \rightarrow_{cbv} e_b[\tau_a/\alpha]
\]

\[
e_f \rightarrow_{cbv} e'_f
\]

\[
e_f [\tau_a] \rightarrow_{cbv} e'_f [\tau_a]
\]

▶ Two new rules. (Note: \( \Lambda \alpha. e_b \) is a value.)

▶ Two new kinds of substitution:
  ▶ \( e[\tau'/\alpha] \) — substitute type for type variable in term (new)
  ▶ \( \tau[\tau'/\alpha] \) — substitute type for type variable in type (new)
Example

\[(\Lambda \alpha. \Lambda \beta. \lambda x: \alpha. \lambda f: \alpha \rightarrow \beta. f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda z: \text{int}. \ z + z)\]
Example

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

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

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

\[\to_{\text{cbv}} (\Lambda \beta. \lambda x: \text{int.} \ \lambda f: \text{int} \to \beta. \ f \ x) \ [\text{int}] \ 3 \ (\lambda z: \text{int.} \ z + z) \]

\[\to_{\text{cbv}} (\lambda x: \text{int.} \ \lambda f: \text{int} \to \text{int.} \ f \ x) \ 3 \ (\lambda z: \text{int.} \ z + z) \]
Example

\[(\Lambda \alpha. \Lambda \beta. \lambda x: \alpha. \lambda f: \alpha \rightarrow \beta. f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda z: \text{int}. z + z)\]

\[\rightarrow_{\text{cbv}} (\Lambda \beta. \lambda x: \text{int}. \lambda f: \text{int} \rightarrow \beta. f \ x) \ [\text{int}] \ 3 \ (\lambda z: \text{int}. z + z)\]

\[\rightarrow_{\text{cbv}} (\lambda x: \text{int}. \lambda f: \text{int} \rightarrow \text{int}. f \ x) \ 3 \ (\lambda z: \text{int}. z + z)\]

\[\rightarrow_{\text{cbv}} (\lambda f: \text{int} \rightarrow \text{int}. f \ 3) \ (\lambda z: \text{int}. z + z)\]
Example

\((\Lambda \alpha. \Lambda \beta. \lambda x: \alpha. \lambda f: \alpha \rightarrow \beta. f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda z: \text{int.} \ z + z)\)

\(\rightarrow_{\text{cbv}} (\Lambda \beta. \lambda x: \text{int.} \ \lambda f: \text{int} \rightarrow \beta. f \ x) \ [\text{int}] \ 3 \ (\lambda z: \text{int.} \ z + z)\)

\(\rightarrow_{\text{cbv}} (\lambda x: \text{int.} \ \lambda f: \text{int} \rightarrow \text{int.} \ f \ x) \ 3 \ (\lambda z: \text{int.} \ z + z)\)

\(\rightarrow_{\text{cbv}} (\lambda f: \text{int} \rightarrow \text{int.} \ f \ 3) \ (\lambda z: \text{int.} \ z + z)\)

\(\rightarrow_{\text{cbv}} (\lambda z: \text{int.} \ z + z) \ 3\)
Example

\((\Lambda \alpha. \Lambda \beta. \lambda x: \alpha. \lambda f: \alpha \rightarrow \beta. f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda z: \text{int}. \ z + z)\)

\[\rightarrow_{\text{cbv}} \ (\Lambda \beta. \lambda x: \text{int}. \lambda f: \text{int} \rightarrow \beta. f \ x) \ [\text{int}] \ 3 \ (\lambda z: \text{int}. \ z + z)\]

\[\rightarrow_{\text{cbv}} \ (\lambda x: \text{int}. \lambda f: \text{int} \rightarrow \text{int}. f \ x) \ 3 \ (\lambda z: \text{int}. \ z + z)\]

\[\rightarrow_{\text{cbv}} \ (\lambda f: \text{int} \rightarrow \text{int}. f \ 3) \ (\lambda z: \text{int}. \ z + z)\]

\[\rightarrow_{\text{cbv}} \ (\lambda z: \text{int}. \ z + z) \ 3\]

\[\rightarrow_{\text{cbv}} \ 3 + 3\]
Example

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

$$\xrightarrow{\text{cbv}} (\Lambda \beta. \lambda x: \text{int.} \ \lambda f: \text{int} \rightarrow \beta. f \ x) \ [\text{int}] \ 3 \ (\lambda z: \text{int.} \ z + z)$$

$$\xrightarrow{\text{cbv}} (\lambda x: \text{int.} \ \lambda f: \text{int} \rightarrow \text{int.} \ f \ x) \ 3 \ (\lambda z: \text{int.} \ z + z)$$

$$\xrightarrow{\text{cbv}} (\lambda f: \text{int} \rightarrow \text{int.} \ f \ 3) \ (\lambda z: \text{int.} \ z + z)$$

$$\xrightarrow{\text{cbv}} (\lambda z: \text{int.} \ z + z) \ 3$$

$$\xrightarrow{\text{cbv}} 3 + 3$$

$$\xrightarrow{\text{cbv}} 6$$
Type System, part 1

Be careful about “no free type variables”:

- Type-checking judgement has the form $\Delta; \Gamma \vdash e : \tau$
  (whole program type-checked with $\cdot; \cdot \vdash e : \tau$)
- Uses a helper “well-formed type” judgement of the form $\Delta \vdash \tau$.
  - “all free type variables of $\tau$ are in $\Delta$”

$\Delta \vdash \tau$
Type System, part 1

Be careful about “no free type variables”:

- Type-checking judgement has the form $\Delta; \Gamma \vdash e : \tau$
  (whole program type-checked with $\cdot; \cdot \vdash e : \tau$)
- Uses a helper “well-formed type” judgement of the form $\Delta \vdash \tau$.
  - “all free type variables of $\tau$ are in $\Delta$”

\[ \Delta \vdash \tau \]

\[
\frac{\Delta \vdash \tau_a \quad \Delta \vdash \tau_r}{\Delta \vdash \tau_a \to \tau_r}
\]

\[
\frac{\alpha \in \Delta}{\Delta \vdash \alpha}
\]

\[
\frac{\Delta \vdash \alpha}{\Delta, \alpha \vdash \tau_r}
\]

Rules are boring (but allowing free type variables is a pernicious source of language/compiler bugs).
Type System, part 2

In the contexts $\Delta$ and $\Gamma$ the expression $e$ has type $\tau$:

$$\Delta; \Gamma \vdash e : \tau$$
Type System, part 2

In the contexts $\Delta$ and $\Gamma$ the expression $e$ has type $\tau$:

$$\Delta; \Gamma \vdash e : \tau$$

$$\Delta; \Gamma \vdash c : \text{int}$$

$$\Gamma(x) = \tau \quad \Delta; \Gamma \vdash x : \tau$$

$$\Delta \vdash \tau_a \quad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r$$

$$\Delta; \Gamma \vdash \lambda x : \tau_a. \ e_b : \tau_a \rightarrow \tau_r$$

$$\Delta; \Gamma \vdash e_f : \tau_a \rightarrow \tau_r \quad \Delta; \Gamma \vdash e_a : \tau_a$$

$$\Delta; \Gamma \vdash e_f \ e_a : \tau_r$$

$$\Delta, \alpha; \Gamma \vdash e_b : \tau_r$$

$$\Delta; \Gamma \vdash \Lambda \alpha. \ e_b : \forall \alpha. \ \tau_r$$

$$\Delta; \Gamma \vdash e_f : \forall \alpha. \ \tau_r \quad \Delta \vdash \tau_a$$

$$\Delta; \Gamma \vdash e_f [\tau_a] : \tau_r[\tau_a/\alpha]$$

iad

One technical change to an old rule.

Two new rules.

One new kind of substitution:

$\tau[\tau'/\alpha]$ — substitute type for type variable in type (new)
Example

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

The typing derivation is rather tall and painful, but just a syntax-directed derivation by instantiating the typing rules.
System F

\[
e ::= \ c \mid x \mid \lambda x: \tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e \ [\tau]
\]

\[
\tau ::= \ \text{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha. \ \tau
\]

\[
v ::= \ c \mid \lambda x: \tau. \ e \mid \Lambda \alpha. \ e
\]

\[
\Gamma ::= \ \cdot \mid \Gamma, x: \tau
\]

\[
\Delta ::= \ \cdot \mid \Delta, \alpha
\]

\[
e \to_{cbv} e'
\]

\[
(\lambda x: \tau. \ e_b) \ v_a \to_{cbv} e_b[v_a/x]
\]

\[
e_f \to_{cbv} e'_f
\]

\[
e_f \ a \to_{cbv} e'_f \ a
\]

\[
\Lambda \alpha. \ e_b \to_{cbv} e'_f \ [\tau_a]
\]

\[
\Delta \vdash \tau_a \to \tau_r
\]

\[
\Delta \vdash \alpha
\]

\[
\Delta, \alpha \vdash \tau_r
\]

\[
\Delta; \Gamma \vdash e : \tau
\]

\[
\Gamma(x) = \tau
\]

\[
\Delta; \Gamma \vdash x : \tau
\]

\[
\Delta; \Gamma \vdash \lambda x : \tau_a. \ e_b : \tau_a \to \tau_r
\]

\[
\Delta; \Gamma \vdash e_f : \tau_a \to \tau_r
\]

\[
\Delta; \Gamma \vdash e_f \ a : \tau_r
\]

\[
\Delta; \Gamma \vdash \forall \alpha. \ \tau_r
\]

\[
\Delta; \Gamma \vdash \tau_a
\]

\[
\Delta; \Gamma \vdash [\tau_a] : \tau_r[\tau_a/\alpha]
\]
Examples

An overly simple polymorphic function...

Let \( \text{id} = \Lambda \alpha . \lambda x : \alpha . x \)

- \( \text{id} \) has type \( \forall \alpha . \alpha \to \alpha \) (and also \( \forall \beta . \beta \to \beta \) and also ...)
- \( \text{id} [\text{int}] \) has type \( \text{int} \to \text{int} \)
- \( \text{id} [\text{int} \ast \text{int}] \) has type \( (\text{int} \ast \text{int}) \to (\text{int} \ast \text{int}) \)
- \( (\text{id} [\forall \beta . \beta \to \beta]) \) \( \text{id} \) has type \( \forall \beta . \beta \to \beta \)

In SML you can't do the last one; in System F you can.
More Examples

Let \( \text{apply1} = \Lambda \alpha. \Lambda \beta. \lambda x: \alpha. \lambda f: \alpha \rightarrow \beta. f \; x \)

- has type \( \forall \alpha. \forall \beta. \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \)
- \( \cdot; g: \text{int} \rightarrow \text{int} \vdash (\text{apply1} \; [\text{int}] \; [\text{int}] \; 3 \; g) : \text{int} \)

Let \( \text{apply2} = \Lambda \alpha. \lambda x: \alpha. \Lambda \beta. \lambda f: \alpha \rightarrow \beta. f \; x \)

- has type \( \forall \alpha. \alpha \rightarrow (\forall \beta. (\alpha \rightarrow \beta) \rightarrow \beta) \)
- \( \cdot; g: \text{int} \rightarrow \text{string}, h: \text{int} \rightarrow \text{int} \vdash \left( \begin{aligned} &\text{let } z = \text{apply2} \; [\text{int}] \; 3 \text{ in} \\ & (z \; [\text{int}] \; h, z \; [\text{string}] \; g) \end{aligned} \right) : \text{int} \ast \text{string} \)

Let \( \text{twice} = \Lambda \alpha. \lambda x: \alpha. \lambda f: \alpha \rightarrow \alpha. f \; (f \; x) \).

- has type \( \forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \)
- Cannot be made more polymorphic
Looking back, looking forward

Have defined System F.

Next:
- 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

▶ Soundness: System F is type-safe
  ▶ (need a Type Substitution Lemma)
▶ Termination: All programs terminate
  ▶ (shocking, since some “self-applications” allowed (id [\forall \alpha. \alpha \to \alpha] \ id))
▶ Parametricity, a.k.a. theorems for free
  ▶ Example:
    If \*; \* \vdash e : \forall \alpha. \forall \beta. (\alpha \ast \beta) \to (\beta \ast \alpha),
    then e is equivalent to \Lambda \alpha. \Lambda \beta. \lambda x: \alpha \ast \beta. (x.2, x.1).
    Every term with this type is the swap function!!
    Intuition:
    e has no way to make an \alpha or a \beta and it cannot tell what \alpha or \beta are
    (or raise an exception or diverge or . . . )
▶ Erasure: Types do not affect run-time behavior

Note: Mutation “breaks everything”
Application of Polymorphism: Security from Safety?

Example: A process $e$ should not access files it did not open.

Require an untrusted process $e$ to type check as follows:

\[
\cdot; \cdot \vdash e : \forall \alpha. \{\text{fopen} : \text{string} \to \alpha, \text{fread} : \alpha \to \text{int}\} \to \text{unit}
\]

This type ensures that the process won’t “forge a file handle” and pass it to fread.

Therefore:

- fread doesn’t need to check (faster)
- file handles don’t need to be encrypted (safer)
- ...
Application of Polymorphism: Security from Safety?

In the Simply-Typed LC, type safety just means not getting stuck.

With type abstraction, it enables secure interfaces!

Suppose file-handles are implemented by \texttt{ints}.
Instantiate $\alpha$ with \texttt{int}, but untrusted code \textit{cannot tell}.

Type safety (and memory safety) is a necessary but insufficient condition for language-based \textit{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 type-checking, we changed:

- the syntax via $\Lambda \alpha. \, e$ and $e[\tau]$
- the operational semantics via type substitution
- the type system via $\Delta$

The operational semantics has not “really” changed; types need not exist at run-time.

Formally:
There is a translation from System F to the untyped lambda-calculus that erases all types and yields an equivalent program:

$$e \rightarrow_{cbv} e' \text{ in System F iff } E[e] \rightarrow_{cbv} E[e'] \text{ in the untyped lambda calculus.}$$

“Erasure and evaluation commute.”
Erasure

The erasure translation is easy to define:

\[
\begin{align*}
\mathcal{E}[c] &= c \\
\mathcal{E}[x] &= x \\
\mathcal{E}[\lambda x : \tau. \ e] &= \lambda x. \mathcal{E}[e] \\
\mathcal{E}[e_1 \ e_2] &= \mathcal{E}[e_1] \mathcal{E}[e_2] \\
\mathcal{E}[\Lambda \alpha. \ e] &= \lambda _. \mathcal{E}[e] \\
\mathcal{E}[e \ [\tau]] &= \mathcal{E}[e] 0
\end{align*}
\]

In pure System F, preserving evaluation order isn’t crucial, but it is with fix, exceptions, mutation, etc.
Connection to real programming languages

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 is a little different. In fact, it is an implicitly typed restriction of System F.

These two qualifications (“implicit” and “restriction”) are deeply related.
Restrictions

- All types have the form $\forall \alpha_1, \ldots, \alpha_n. \tau$ where $n \geq 0$ and $\tau$ has no $\forall$.
  - (Prenex-quantification; no first-class polymorphism.)
- Only `val` (and `fun`) variables (e.g., `x` in `val x = e`) can have polymorphic types.
  - (Let-bound polymorphism)
- Monomorphic types ($n = 0$) for other variables:
  - function arguments, pattern variables, etc.
  - Cannot (always) desugar `let` to `\lambda` in ML.
- In `fun f x = e`, the variable $f$ can have type $\forall \alpha_1, \ldots, \alpha_n. \tau_a \rightarrow \tau_r$ only if every use of $f$ in $e$ instantiates each $\alpha_i$ with $\alpha_i$.
  - (No polymorphic recursion)
- `val` variables can be polymorphic only if $e$ is a “syntactic value”
  - a variable, constant, function definition, ...
  - (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 is undecidable (1995).
▶ (given untyped $e$, is there a System F term $e'$ such that $E[e'] = e$?)

▶ 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 mutable references, but the value-restriction restores soundness.
▶ unification algorithm
Recovering lost ground?

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

- require type annotations
- 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?
Type Inference for First-Class Polymorphism

- Type reconstruction with first-class polymorphic values;
  J. W. O’Toole, Jr., D. K. Gifford; PLDI’89
- Putting type annotations to work;
  Martin Odersky, Konstantin Läufer; POPL’96
- First-class polymorphism with type inference;
  Mark Jones; POPL’97
- Semi-explicit first-class polymorphism for ML;
  Jacques Garrigue, Didier Rémy; I&C v.155-n.1/2 Nov/Dec’99
- MLF: raising ML to the power of System F;
  Didier Le Botlan, Didier Rémy; ICFP’03
- Qualified types for MLF;
  Daan Leijen, Andres Löh; ICFP’05
- Simple, partial type-inference for System F based on type-containment;
  Didier Rémy; ICFP’05
- Boxy types: inference for higher-rank types and impredicativity;
  Dimitrios Vytiniotis, Stephanie Weirich, Simon Peyton Jones; ICFP’06
- Practical type inference for arbitrary-rank types;
  Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, Mark Shields; JFP v.17-n.1 Jan’07
- HMF: simple type inference for first-class polymorphism;
  Daan Leijen; ICFP’08
- FPH: first-class polymorphism for Haskell;
  Dimitrios Vytiniotis, Stephanie Weirich, Simon Peyton Jones; ICFP’08
- Flexible types: robust type inference for first-class polymorphism;
  Daan Leijen; POPL’09
- QML: explicit first-class polymorphism for ML;
  Claudio V. Russo, Dimitrios Vytiniotis; ML’09
- Complete and easy bidirectional typechecking for higher-rank polymorphism;
  Joshua Dunfield, Neelakantan R. Krishnaswami; ICFP’13
- Guarded impredicative polymorphism;
  Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones; PLDI’18