FPH: First-class polymorphism for Haskell

Stephanie Weirich
joint work with Dimitrios Vytiniotis and Simon Peyton Jones

Computer and Information Science Department
University of Pennsylvania

Park City UT, June 2008

Unleashing polymorphism

- First-class functions are good

Unleashing polymorphism

- First-class functions are good
- Polymorphic functions are good

Unleashing polymorphism

- First-class functions are good
- Polymorphic functions are good
- But where are the first-class polymorphic functions?

Unleashing polymorphism

- First-class functions are good
- Polymorphic functions are good
- But where are the first-class polymorphic functions?

```
g :: (forall a. a -> a -> a) -> (Bool, Int)
g sel = (sel True False, sel 1 2)
```

Unleashing polymorphism

- First-class functions are good
- Polymorphic functions are good
- But where are the first-class polymorphic functions?

```
g :: (forall a. a -> a -> a) -> (Bool, Int)
g sel = (sel True False, sel 1 2)
```

```
f :: [forall a. a -> a -> a] -> (Bool, Int)
f sels = ((head sel) True False, (head sel) 1 2)
```

This talk: extending Damas-Milner type inference to support rich polymorphism

Why no first-class polymorphism?

Damas-Milner has two expressiveness restrictions

1. $\forall$ quantifiers allowed only at top-level
   - eg: $[\forall a.a \to a \to a] \to (Bool, Int)$ not allowed
   - Damas-Milner types: $\forall a_1 \ldots \forall a_n.\tau$ where τ is quantifier-free
   - Rich types: contain arbitrary polymorphism
2. Instantiations only with quantifier-free types:
   - eg: `head sels` not allowed, even if `sels` : $[\forall a.a \to a \to a]$

Lifting restriction [1]: arbitrary-rank types

Arbitrary-rank types: arbitrary polymorphism under "$\rightarrow$"

```
f get = (get 3, get False)
```

Lifting restriction [1]: arbitrary-rank types

Arbitrary-rank types: arbitrary polymorphism under "$\rightarrow$"

```
f get = (get 3, get False)
```

Many possible types for `f`:

- $(\forall a.a \rightarrow Int) \rightarrow (Int, Int)$
- $(\forall a.a \rightarrow a) \rightarrow (Int, Bool)$

Lifting restriction [1]: arbitrary-rank types

Arbitrary-rank types: arbitrary polymorphism under "$\rightarrow$"

```
f get = (get 3, get False)
```

Many possible types for `f`:

- $(\forall a.a \rightarrow Int) \rightarrow (Int, Int)$
- $(\forall a.a \rightarrow a) \rightarrow (Int, Bool)$

No principal type, no single one to choose and use throughout the scope of the definition

$\Longrightarrow$ modular type inference: impossible

Arbitrary-rank types: problem solved, really

[Odersky & Läufer, 1996]

```
f (get :: forall a.a -> a) = (get 3, get False)
```

Key ideas

- Exploit type annotations for arbitrary-rank type inference
- Annotate function arguments that must be polymorphic

[Peyton Jones, Vytiniotis, Weirich, Shields, 2007]

- Propagation of type annotations to basic O-L
- Fewer annotations, better error messages
- Explored further metatheory and expressiveness

Example: Haskell generic programming

Scrap your boilerplate [Lämmel & Peyton Jones, 2003]

```
class Typeable a => Data a where
    ...
    gmapT :: (forall b.Data b => b -> b) -> a -> a
    gmapQ :: (forall a.Data a => a -> u) -> a -> [u]
    ...
```

`gmapT` applies a transformation to immediate subnodes in a data structure independently of what type these subnodes have, as long as they are instances of `Data`

Example: Haskell generic programming

Scrap your boilerplate [Lämmel & Peyton Jones, 2003]

```
class Typeable a => Data a where
    ...
    gmapT :: (forall b.Data b => b -> b) -> a -> a
    gmapQ :: (forall a.Data a => a -> u) -> a -> [u]
    ...
```

gmapT applies a transformation to immediate subnodes in a data structure independently of what type these subnodes have, as long as they are instances of Data

Example: Encapsulating state, purely functionally

State transformers for Haskell [Peyton Jones & Launchbury, 1994]

```
data ST s a
data STRef s a
newSTRef :: forall s a.a -> ST s (STRef s a)

runST :: forall a.(forall s.ST s a) -> a
```

Example: Encapsulating state, purely functionally

State transformers for Haskell [Peyton Jones & Launchbury, 1994]

```
data ST s a
data STRef s a
newSTRef :: forall s a.a -> ST s (STRef s a)

runST :: forall a.(forall s.ST s a) -> a
```

runST encapsulates stateful computation and returns a pure result.

Type prevents state to "escape" the encapsulation

```
let v = runST (newSTRef True) in ...  -- should fail!
```

Lifting restriction [2]: impredicative instantiations

```
runST :: forall a.(forall s.ST s a) -> a
($) :: forall a b.(a -> b) -> a -> b

f = runST $ arg
```

Must instantiate `a` of $ with `forall s.ST s ...`

Problematic for type inference, again

```
choose :: forall a.a -> a -> a
id :: forall b.b -> b

goo = choose id
```

$$
\begin{array}{lcl}
\mathsf{a} \mapsto (b \to b) & \Longrightarrow & \mathsf{goo} : \forall b.(b \to b) \to b \to b \\
\mathsf{a} \mapsto (\forall b.b \to b) & \Longrightarrow & \mathsf{goo} : (\forall b.b \to b) \to (\forall b.b \to b)
\end{array}
$$

Problematic for type inference, again

```
choose :: forall a.a -> a -> a
id :: forall b.b -> b

goo = choose id
```

$$
\begin{array}{lcl}
\mathsf{a} \mapsto (b \to b) & \Longrightarrow & \mathsf{goo} : \forall b.(b \to b) \to b \to b \\
\mathsf{a} \mapsto (\forall b.b \to b) & \Longrightarrow & \mathsf{goo} : (\forall b.b \to b) \to (\forall b.b \to b)
\end{array}
$$

Incomparable types for definitions. Which one to choose?

- No principal types $\Longrightarrow$ no modular type inference

“Local” type inference tempting but not satisfactory

We may try to use type annotation propagation to let the type checker decide about instantiations locally. Difficult to make it work

```
length :: forall a.[a] -> Int
ids :: [forall a.a->a]
f :: [forall b.b->b] -> Int
[] :: forall a.[a]

h0 = length ids
h1 = f []

h2 :: [forall a.a -> a]
h2 = cons (λx.x) []

h3 = cons (λx.x) (reverse ids)
```

Who determines polymorphic instantiation in applications?

- Argument type?
- Function type?
- Type annotation?
- Some subexpression?

“Local” type inference tempting but not satisfactory

We may try to use type annotation propagation to let the type checker decide about instantiations locally. Difficult to make it work

```
length :: forall a.[a] -> Int
ids :: [forall a.a->a]
f :: [forall b.b->b] -> Int
[] :: forall a.[a]

h0 = length ids
h1 = f []

h2 :: [forall a.a -> a]
h2 = cons (λx.x) []

h3 = cons (λx.x) (reverse ids)
```

Who determines polymorphic instantiation in applications?

- Argument type?
- Function type?
- Type annotation?
- Some subexpression?

"Local" type inference tempting but not satisfactory

We may try to use type annotation propagation to let the type checker decide about instantiations locally. Difficult to make it work

```
length :: forall a.[a] -> Int
ids :: [forall a.a->a]
f :: [forall b.b->b] -> Int
[] :: forall a.[a]

h0 = length ids
h1 = f []

h2 :: [forall a.a -> a]
h2 = cons (λx.x) []

h3 = cons (λx.x) (reverse ids)
```

Who determines polymorphic instantiation in applications?

- Argument type?
- Function type?
- Type annotation?
- Some subexpression?

“Local” type inference tempting but not satisfactory

We may try to use type annotation propagation to let the type checker decide about instantiations locally. Difficult to make it work

```
length :: forall a.[a] -> Int
ids :: [forall a.a->a]
f :: [forall b.b->b] -> Int
[] :: forall a.[a]

h0 = length ids
h1 = f []

h2 :: [forall a.a -> a]
h2 = cons (λx.x) []

h3 = cons (λx.x) (reverse ids)
```

Who determines polymorphic instantiation in applications?

- Argument type?
- Function type?
- Type annotation?
- Some subexpression?

“Local” type inference tempting but not satisfactory

We may try to use type annotation propagation to let the type checker decide about instantiations locally. Difficult to make it work

```
length :: forall a.[a] -> Int
ids :: [forall a.a->a]
f :: [forall b.b->b] -> Int
[] :: forall a.[a]

h0 = length ids
h1 = f []

h2 :: [forall a.a -> a]
h2 = cons (λx.x) []

h3 = cons (λx.x) (reverse ids)
```

Who determines polymorphic instantiation in applications?

- Argument type?
- Function type?
- Type annotation?
- Some subexpression?

Back to the drawing board

```
choose :: forall a.a -> a -> a
id :: forall b.b -> b

goo = choose id
```

$$\begin{array}{lcl} \mathsf{a} \mapsto (b \to b) & \Longrightarrow & \mathsf{goo} : \forall b.(b \to b) \to b \to b \\ \mathsf{a} \mapsto (\forall b.b \to b) & \Longrightarrow & \mathsf{goo} : (\forall b.b \to b) \to (\forall b.b \to b) \end{array}$$

Back to the drawing board

```
choose :: forall a.a -> a -> a
id :: forall b.b -> b

goo = choose id
```

$$
\begin{array}{lcl}
\mathsf{a} \mapsto (b \to b) & \Longrightarrow & \mathsf{goo} : \forall b.(b \to b) \to b \to b \\
\mathsf{a} \mapsto (\forall b.b \to b) & \Longrightarrow & \mathsf{goo} : (\forall b.b \to b) \to (\forall b.b \to b)
\end{array}
$$

Problem can be fixed if we go beyond System F:

- The type

  $$\forall a \geq (\forall b.b \to b).a \to a$$

  is the principal (non System F) type for `choose id`

The $\mathrm{ML^F}$ solution

The $\mathrm{ML^F}$ solution [Le Botlan & Rémy, 2003]: extend the type language beyond System F to recover principal types

- Expose constraints in the high-level specification
- Algorithm manipulates instantiation constraints
- Substantial additional machinery in the specification compared to Damas-Milner
- But expressive, robust, requires a small number of type annotations

The ML^F solution

The ML^F solution [Le Botlan & Rémy, 2003]: extend the type language beyond System F to recover principal types

- Expose constraints in the high-level specification
- Algorithm manipulates instantiation constraints
- Substantial additional machinery in the specification compared to Damas-Milner
- But expressive, robust, requires a small number of type annotations

ML^F: the main inspiration for this work

A Challenge Problem for 2003-2008

- Can we achieve a constraint-free specification of type inference for first-class polymorphism?

A Challenge Problem for 2003-2008

- Can we achieve a constraint-free specification of type inference for first-class polymorphism?
- Want simplicity, expressiveness, robustness, backwards compatibility, . . .

A Challenge Problem for 2003-2008

- Can we achieve a constraint-free specification of type inference for first-class polymorphism?
- Want simplicity, expressiveness, robustness, backwards compatibility, . . .
- Can we give clear guidelines to programmers about where type annotations are needed?

A principled "global" approach

Look again at where, in theory, you run into trouble with ambiguity

A principled “global” approach

Look again at where, in theory, you run into trouble with ambiguity

- Theory says: “let-bound definitions and abstractions” because there you must choose which type to use!
- Intuition: “if Theory is correct, these should be the ONLY places where you may need annotations”

Can we make this work?

A principled "global" approach

Look again at where, in theory, you run into trouble with ambiguity

- Theory says: "let-bound definitions and abstractions" because there you must choose which type to use!
- Intuition: "if Theory is correct, these should be the ONLY places where you may need annotations"

Can we make this work?

No "local" decisions: postpone instantiation decisions using constraints in the algorithm, until forced to make a decision

A principled “global” approach

Look again at where, in theory, you run into trouble with ambiguity

- Theory says: “let-bound definitions and abstractions” because there you must choose which type to use!
- Intuition: “if Theory is correct, these should be the ONLY places where you may need annotations”

Can we make this work?

No “local” decisions: postpone instantiation decisions using constraints in the algorithm, until forced to make a decision

But never expose these constraints in your specification, pretend you knew the solution to the constraints from the beginning

The running example

```
choose :: forall a.a -> a -> a
id :: forall b.b->b

goo = choose id
```

Suppose we allow in the type system any instantiation of `choose`:

1. $\mathsf{a} \mapsto (b \to b) \quad \Longrightarrow \quad \mathsf{goo} : \forall b.(b \to b) \to b \to b$
2. $\mathsf{a} \mapsto (\forall b.b \to b) \quad \Longrightarrow \quad \mathsf{goo} : (\forall b.b \to b) \to (\forall b.b \to b)$

The running example

```
choose :: forall a.a -> a -> a
id :: forall b.b->b

goo = choose id
```

Suppose we allow in the type system any instantiation of `choose`:

1. $\mathsf{a} \mapsto (b \to b) \quad \Longrightarrow \quad \mathsf{goo} : \forall b.(b \to b) \to b \to b$
2. $\mathsf{a} \mapsto (\forall b.b \to b) \quad \Longrightarrow \quad \mathsf{goo} : (\forall b.b \to b) \to (\forall b.b \to b)$

At a definition point, we have to decide which one we want

The running example

```
choose :: forall a.a -> a -> a
id :: forall b.b->b

goo = choose id
```

Suppose we allow in the type system any instantiation of `choose`:

1. $\mathsf{a} \mapsto (b \to b) \quad \Longrightarrow \quad \mathsf{goo} : \forall b.(b \to b) \to b \to b$
2. $\mathsf{a} \mapsto (\forall b.b \to b) \quad \Longrightarrow \quad \mathsf{goo} : (\forall b.b \to b) \to (\forall b.b \to b)$

At a definition point, we have to decide which one we want

Key insight: Type system keeps track of ambiguity in the types.
At a let-definition the type of the expression to be bound must be unambiguous

Boxes around ambiguous types

- Use a special type constructor $\boxed{\sigma}$
- Call it a box. It "guards" impredicative instantiations
- Instantiate with boxy monomorphic types τ:

$$\begin{array}{rcl} \tau & ::= & a \mid \tau \to \tau \mid [\tau] \mid \boxed{\sigma} \\ \sigma & ::= & \forall a.\sigma \mid a \mid \sigma \to \sigma \mid [\sigma] \end{array}$$

τ-types: ordinary Damas-Milner quantifier-free types + boxy types

Instantiation exactly as in Damas-Milner

$$\frac{(\texttt{choose} : \forall a.a \to a \to a) \in \Gamma}{\Gamma \vdash \texttt{choose} : \tau \to \tau \to \tau}$$

for any τ!

Typing (`choose id`) in the specification

First way (as in Damas-Milner)

1. Instantiate `choose` : $(b \to b) \to (b \to b) \to b \to b$
2. Instantiate `id` to $b \to b$
3. Match-up type of `id` with the type that `choose` requires

$$b \to b \equiv b \to b$$

4. Result is `choose id` : $(b \to b) \to b \to b$

Typing `(choose id)` in the specification

First way (as in Damas-Milner)

1. Instantiate `choose` : $(b \to b) \to (b \to b) \to b \to b$
2. Instantiate `id` to $b \to b$
3. Match-up type of `id` with the type that `choose` requires

$$b \to b \equiv b \to b$$

4. Result is `choose id` : $(b \to b) \to b \to b$

Second way

1. Instantiate `choose` : $\boxed{\forall b.b \to b} \to \boxed{\forall b.b \to b} \to \boxed{\forall b.b \to b}$
2. Match-up type of `id` with the type that `choose` requires ignoring boxes

$$\forall b.b \to b \equiv \boxed{\forall b.b \to b}$$

3. Result is `choose id` : $\boxed{\forall b.b \to b} \to \boxed{\forall b.b \to b}$

Again multiple types for `(choose id)`?

Indeed:

$$\texttt{choose id} : \boxed{\forall b.b \to b} \to \boxed{\forall b.b \to b}$$
$$\texttt{choose id} : \forall b.(b \to b) \to b \to b$$

A boxy type is a warning for ambiguity!

Let-bound definitions must have box-free types $\Longrightarrow$ no ambiguity

$$\frac{\Gamma \vdash u : \sigma \quad \sigma \text{ is box-free} \quad \Gamma, (x{:}\sigma) \vdash e : \sigma'}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e : \sigma'}\ \text{let}$$

Hence, can only bind goo with type $\forall b.(b \to b) \to b \to b$

Again multiple types for `(choose id)`?

Indeed:

$$\texttt{choose id} : \boxed{\forall b.b \to b} \to \boxed{\forall b.b \to b}$$

$$\texttt{choose id} : \forall b.(b \to b) \to b \to b$$

A boxy type is a warning for ambiguity!

Let-bound definitions must have box-free types $\Longrightarrow$ no ambiguity

$$\frac{\Gamma \vdash u : \sigma \quad \sigma \text{ is box-free} \quad \Gamma, (x{:}\sigma) \vdash e : \sigma'}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e : \sigma'}\ \text{let}$$

Hence, can only bind goo with type $\forall b.(b \to b) \to b \to b$

As in Damas-Milner: backwards compatibility

Annotations recover other types

```
goo :: (forall b.b->b) -> (forall b.b->b)
goo = choose id
```

Same idea as in applications

1. Type `choose id` : $\boxed{\forall b.b \to b} \to \boxed{\forall b.b \to b}$
2. Match-up annotation with expression type ignoring boxes

$$\boxed{\forall b.b \to b} \to \boxed{\forall b.b \to b} \equiv (\forall b.b \to b) \to \forall b.b \to b$$

3. Bind goo with the box-free type from the annotation

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \equiv \sigma}{\Gamma \vdash (e{::}\sigma) : \sigma}\ \textsf{ann}$$

Boxes do not get in the way

Boxes only used at lets.

```
head :: forall a.[a] -> a

g = head ids 42
```

The boxy type of `head ids` can automatically become box-free

$$\Gamma \vdash \texttt{head ids} : \boxed{\forall a.a \to a}$$

Boxes do not get in the way

Boxes only used at lets.

```
head :: forall a.[a] -> a

g = head ids 42
```

The boxy type of `head ids` can automatically become box-free

$$\Gamma \vdash \texttt{head ids} : \boxed{\forall a.a \to a} \preceq \boxed{Int \to Int}$$

Relation $\preceq$ instantiates inside boxes

Boxes do not get in the way

Boxes only used at lets.

```
head :: forall a.[a] -> a

g = head ids 42
```

The boxy type of `head ids` can automatically become box-free

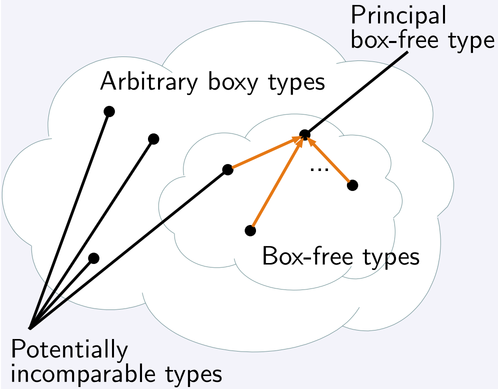
$$\Gamma \vdash \texttt{head ids} : \boxed{\forall a.a \rightarrow a} \preceq \boxed{Int \rightarrow Int} \sqsubseteq Int \rightarrow Int$$

Relation $\preceq$ instantiates inside boxes

Relation $\sqsubseteq$ removes boxes when their contents are monomorphic

What is going on at let-bound definitions?

Many types for the same expression



- No annotation present: a box-free type is chosen
- Other types recovered through annotations

Where does one need an annotation?

If we want to be conservative, we need not think about boxes at all

Guideline: You need only annotate all those definitions (and anonymous functions) that must be typed with rich types

Where does one need an annotation?

If we want to be conservative, we need not think about boxes at all

Guideline: You need only annotate all those definitions (and anonymous functions) that must be typed with rich types

Theorem: If $\Gamma \vdash^{\mathsf{F}} e : \sigma$ (i.e. typeable in implicit System F) and e consists only of constants, variables, and applications, then $\Gamma \vdash e : \sigma'$ such that $\sigma' \equiv \sigma$.

Consequence: very robust for polymorphic combinators such as $

Guideline is conservative: that's why boxes are there!

```
f :: [forall b.b -> b] -> [forall b.b->b]
ids :: [forall b.b->b]
g = f ids
```

Type of `f ids` is a rich type

- But no ambiguity, should need no annotations
- Possible, because type of `f ids` is box-free

Conservativity for a declarative specification

Assume $(\mathtt{h} : \forall a.a \to [a] \to [a]) \in \Gamma$

```
f :: [forall a.a->a] -- annotation required!
f = h id ids
```

Seems silly to require an annotation on `f`

But if `h` gets a more general type, $(\mathtt{h} : \forall ab.a \to [b] \to [a]) \in \Gamma$

```
f = h id ids
```

Then `f` gets the incomparable type $\forall c.[c \to c]$

Conservativity for a declarative specification

Assume $(\mathtt{h} : \forall a.a \to [a] \to [a]) \in \Gamma$

```
f :: [forall a.a->a] -- annotation required!
f = h id ids
```

Seems silly to require an annotation on `f`

But if `h` gets a more general type, $(\mathtt{h} : \forall ab.a \to [b] \to [a]) \in \Gamma$

```
f = h id ids
```

Then `f` gets the incomparable type $\forall c.[c \to c]$

A more general type for a let-bound expression can make a program in its scope untypeable!

Summary

- Simplicity:
  Reminiscent of the declarative Damas-Milner specification
- Expressiveness:
  Can embed all of System F with the addition of annotations to abstractions and let-bindings with rich types
- Robustness:
  Robust in application of polymorphic combinators
- Modularity:
  Principal box-free types for programs
- Backwards compatibility:
  Types all Damas-Milner typeable programs

Summary

- Simplicity:
  Reminiscent of the declarative Damas-Milner specification
- Expressiveness:
  Can embed all of System F with the addition of annotations to abstractions and let-bindings with rich types
- Robustness:
  Robust in application of polymorphic combinators
- Modularity:
  Principal box-free types for programs
- Backwards compatibility:
  Types all Damas-Milner typeable programs

Good candidate for next-generation FP languages

Related work

ML^{F} [Le Botlan & Rémy, 2003]

- Substantial additional machinery in the specification, implementation with constraints, precise guidelines where annotations are needed, fewer annotations are needed

Boxy Types [Vytiniotis *et al.*, 2006]

- Simple implementation, complex syntax-directed spec, little robustness

HM^{F} [Leijen, 2008]

- Simple implementation, somewhat algorithmic specification, harder to tell where annotations are needed

This work

- Simple specification, implementation with constraints, precise annotation guidelines, robust, elegant

Future work

- Interaction with Haskell type classes
- Efficiency considerations
- Implementation in a commercial-scale compiler
- Explore expressiveness improvements
  - Hoisting of $\forall$-quantifiers to the right of $\rightarrow$?

Prototype available!

`www.cis.upenn.edu/~dimitriv/fph`