1 Design
   - $i\text{ML}^F$: an implicity-typed extension of System F
   - Types explained
   - $e\text{ML}^F$: an explicitly-typed version of $i\text{ML}^F$

2 Results
   - Principal types
   - Robustness to program transformations
   - Practice

3 Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in $\text{ML}^F$

4 Concluding remarks
A new look at $\text{ML}^F$

Didier Rémy

INRIA-Rocquencourt

Portland, June 2008

Based on joint work with

Didier Le Botlan and Boris Yakobowski
Simple to use
Expressive
Great success
Happy days
Expressive

Simple extension of ML
Simplification

Even used in full scale languages such as Scala.

Full type inference is undecidable

Full type annotations are obfuscating
Outline

1 Design
   - \textit{iMLF}: an implicity-typed extension of System F
   - Types explained
   - \textit{eMLF}: an explicitly-typed version of \textit{iMLF}

2 Results
   - Principal types
   - Robustness to program transformations
   - Practice

3 Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in MLF

4 Concluding remarks
A universal type system

Explicit System F:

\[ \begin{align*}
\text{VAR} \quad & z : \tau \in \Gamma \quad \Rightarrow \quad \Gamma \vdash z : \tau \\
\text{APP} \quad & \Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash a_2 : \tau_2 \\
\quad & \Gamma \vdash a_1 a_2 : \tau_1 \\
\text{GEN} \quad & \Gamma, \alpha \vdash a : \tau_0 \\
\quad & \Gamma \vdash \concrete{\Lambda \alpha}{a} : \forall (\alpha) \tau_0 \\
\text{UNGEN} \quad & \Gamma \vdash a : \forall (\alpha) \tau \\
\quad & \Gamma \vdash a \, \tau : \tau_0[\alpha \leftarrow \tau] \\
\text{FUN} \quad & \Gamma, x : \tau_0 \vdash a : \tau \\
\quad & \Gamma \vdash \lambda(x : \tau_0) \, a : \tau_0 \rightarrow \tau
\end{align*} \]
A universal type system

**Implicit** System F:

\[
\begin{align*}
\text{VAR} & \\
\frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau} \\
\text{APP} & \\
\frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 \ a_2 : \tau_1} \\
\text{FUN} & \\
\frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x) \ a : \tau_0 \to \tau}
\end{align*}
\]

\[
\begin{align*}
\text{GEN} & \\
\frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall(\alpha) \tau_0} \\
\text{UNGEN} & \\
\frac{\Gamma \vdash a : \forall(\alpha) \tau}{\Gamma \vdash a : \tau_0[\alpha \leftarrow \tau]}
\end{align*}
\]
A universal type system

**Implicit System F:**

\[
\begin{align*}
\text{VAR} & \quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau} \\
\text{APP} & \quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 a_2 : \tau_1} \\
\text{FUN} & \quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda (x) a : \tau_0 \rightarrow \tau} \\
\text{GEN} & \quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall (\alpha) \tau_0} \\
\text{INST} & \quad \frac{\forall (\bar{\alpha}) \tau_0 \leq \tau_0[\bar{\alpha} \leftarrow \bar{\tau}]}{} \\
\text{SUB} & \quad \frac{\Gamma \vdash a : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash a : \tau_2}
\end{align*}
\]
A universal type system

Implicit System F:

\[
\begin{align*}
\text{VAR} & : \quad z : \tau \in \Gamma \quad \Rightarrow \quad \Gamma \vdash z : \tau \\
\text{APP} & : \quad \Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1, \quad \Gamma \vdash a_2 : \tau_2 \quad \Rightarrow \quad \Gamma \vdash a_1 a_2 : \tau_1 \\
\text{FUN} & : \quad \Gamma, x : \tau_0 \vdash a : \tau \quad \Rightarrow \quad \Gamma \vdash \lambda(x) a : \tau_0 \rightarrow \tau \\
\text{GEN} & : \quad \Gamma, \alpha \vdash a : \tau_0 \quad \Rightarrow \quad \Gamma \vdash a : \forall(\alpha) \tau_0 \\
\text{INST} & : \quad \bar{\beta} \notin \text{ftv}(\forall(\bar{\alpha}) \bar{\tau}_0) \quad \Rightarrow \quad \forall(\bar{\alpha}) \tau_0 \leq \forall(\bar{\beta}) \tau_0[\bar{\alpha} \leftarrow \bar{\tau}] \\
\text{SUB} & : \quad \Gamma \vdash a : \tau_1, \quad \tau_1 \leq \tau_2 \quad \Rightarrow \quad \Gamma \vdash a : \tau_2
\end{align*}
\]
A universal type system

**Implicit** System F:

\[
\begin{align*}
\text{VAR} & & \Gamma \vdash z : \tau \\
\text{App} & & \Gamma \vdash a_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash a_2 : \tau_2 \quad \Gamma \vdash a_1 \ a_2 : \tau_1 \\
\text{Fun} & & \Gamma, x : \tau_0 \vdash a : \tau \\
\text{Gen} & & \Gamma, \alpha \vdash a : \tau_0 \quad \Gamma \vdash a : \forall(\alpha) \tau_0 \\
\text{Inst} & & \beta \notin \text{ftv}(\forall(\alpha) \tau_0) \quad \forall(\alpha) \tau_0 \leq \forall(\beta) \tau_0[\alpha \leftarrow \bar{\tau}] \\
\text{Sub} & & \Gamma \vdash a_1 : \tau_1 \quad \tau_1 \leq \tau_2 \quad \Gamma \vdash a : \tau_2 \\
\text{LET} & & \Gamma \vdash a_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash a_2 : \tau \quad \Gamma \vdash \text{let } x = a_1 \ \text{in} \ a_2 : \tau
\end{align*}
\]

Add a construction for local bindings (perhaps derivable):

\[
\Gamma \vdash a_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash a_2 : \tau \quad \Gamma \vdash \text{let } x = a_1 \ \text{in} \ a_2 : \tau
\]
A universal type system

**Implicit System F:**

\[
\begin{array}{c}
\text{VAR} \\
\frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau}
\end{array}
\quad
\begin{array}{c}
\text{APP} \\
\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 \ a_2 : \tau_1 \rightarrow \tau}
\end{array}
\quad
\begin{array}{c}
\text{GEN} \\
\frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash \lambda (x) \ a : \forall (\alpha) \tau_0}
\end{array}
\]

**Logical, canonical presentation of typing rules**

- Covers many variations: \( F, ML, F^n, F_\leq, \ldots \)
  - Vary the set of types.
  - Vary the instance relation between types.
- For ML, just restrict types to ML types.

Add a construction for local bindings (perhaps derivable):

\[
\begin{array}{c}
\text{LET} \\
\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash a_2 : \tau}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau}
\end{array}
\]
A universal type system

**Implicit System F:**

\[ \text{VAR} \]
\[ z : \tau \in \Gamma \]
\[ \Gamma \vdash z : \tau \]

\[ \text{App} \]
\[ \Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \]
\[ \Gamma \vdash a_2 : \tau_2 \]
\[ \Gamma \vdash a_1 a_2 : \tau_1 \]

\[ \text{Fun} \]
\[ \Gamma, x : \tau_0 \vdash a : \tau_0 \]

\[ \text{Gen} \]
\[ \Gamma, \alpha \vdash a : \tau_0 \]
\[ \Gamma \vdash a : \forall (\alpha) \tau_0 \]

**Logical, canonical presentation of typing rules**

- Covers many variations: F, ML, F^\eta, F_\leq, ...
  - Vary the set of types.
  - Vary the instance relation between types.
- For ML, **just restrict** types to ML types.

**Do never change the typing rules!**

Add a construction for local bindings (perhaps derivable):

\[ \text{LET} \]
\[ \Gamma \vdash a_1 : \tau_1 \]
\[ \Gamma, x : \tau_1 \vdash a_2 : \tau \]
\[ \Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau \]
Type inference is undecidable — in System F

Of course, we must

- Use type annotations on function parameters in some cases.

When?

- Always?
  - too many annotations are obfuscating.
- Alleviate some annotations by local type inference?
  - unintuitive and fragile (to program transformations).
- When parameters have polymorphic types?
  - still too many bothersome type annotations.

Are polymorphic types less important than monomorphic ones?
Type inference is undecidable — in System F

Of course, we must

- Use type annotations on function parameters in some cases.

When?

- Always?
  - too many annotations are obfuscating.
- Alleviate some annotations by local type inference?
  - unintuitive and fragile (to program transformations).
- When parameters have polymorphic types?
  - still too many bothersome type annotations.

Are polymorphic types less important than monomorphic ones?

Our choice

- When (and only when) parameters are used polymorphically.
Lack of principal types for applications

The example of choice

let choice = \( \lambda(x) \lambda(y) \) if true then \( x \) else \( y \) : \( \forall \beta \cdot \beta \rightarrow \beta \rightarrow \beta \)

let id = \( \lambda(z) z \) : \( \forall(\alpha) \alpha \rightarrow \alpha \)

choice id :
Lack of principal types for applications

The example of choice

let choice = λ(x) λ(y) if true then x else y : ∀ β · β → β → β
let id = λ(z) z : ∀(α) α → α

\[
\text{choice id} : \begin{cases} \\
\forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\
(\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)
\end{cases}
\]
Lack of principal types for applications

The example of choice

\[
\text{let } \text{choice} = \lambda(x) \lambda(y) \text{ if } \text{true} \text{ then } x \text{ else } y : \forall \beta \cdot \beta \rightarrow \beta \rightarrow \beta \\
\text{let } \text{id} = \lambda(z) z : \forall(\alpha) \alpha \rightarrow \alpha
\]

\[
\text{choice id : } \begin{cases}
\forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\
(\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)
\end{cases} \quad \text{No better choice in F!}
\]
Lack of principal types for applications

The example of choice

\[
\text{let } \text{choice} = \lambda(x) \lambda(y) \text{ if } \text{true} \text{ then } x \text{ else } y : \forall \beta \cdot \beta \rightarrow \beta \rightarrow \beta
\]

\[
\text{let } \text{id} = \lambda(z) z : \forall(\alpha) \alpha \rightarrow \alpha
\]

\[
\text{choice } \text{id} : \begin{cases} 
\forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\
(\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)
\end{cases}
\]

No better choice in F!

The problem is serious and inherent

- Follows from rules \text{INST}, \text{GEN}, and \text{APP}.
- Should values be kept as polymorphic or as instantiated as possible?
- A type inference system \textit{can} do both, but \textit{cannot} choose.
Lack of principal types for applications

The example of choice

let choice = \( \lambda(x) \lambda(y) \text{if} \ \text{true} \ \text{then} \ x \ \text{else} \ y \) : \( \forall \beta \cdot \beta \to \beta \to \beta \)

let id = \( \lambda(z) \ z \) : \( \forall(\alpha) \ \alpha \to \alpha \)

\[
\begin{align*}
\text{choice id : } & \left\{ \begin{array}{l}
\forall(\alpha) (\alpha \to \alpha) \to (\alpha \to \alpha) \\
(\forall(\alpha) \ \alpha \to \alpha) \to (\forall(\alpha) \ \alpha \to \alpha)
\end{array} \right.
\end{align*}
\]

The solution in i\(\text{MLF}^F\):

\[
\begin{align*}
\text{choice id : } & \quad \forall(\beta \geq \forall(\alpha) \ \alpha \to \alpha) \ \beta \to \beta
\end{align*}
\]
Lack of principal types for applications

The example of choice

let choice = \lambda(x) \lambda(y) if true then x else y : \forall \beta \cdot \beta \to \beta \to \beta
let id = \lambda(z) z : \forall(\alpha) \alpha \to \alpha

choice id : \begin{cases} \forall(\alpha) (\alpha \to \alpha) \to (\alpha \to \alpha) \\ (\forall(\alpha) \alpha \to \alpha) \to (\forall(\alpha) \alpha \to \alpha) \end{cases}

The solution in iMLF:

choice id : \forall(\beta \geq \forall(\alpha) \alpha \to \alpha) \beta \to \beta

\ll \begin{cases} (\beta \to \beta) [\beta \leftarrow \forall(\alpha) \alpha \to \alpha] \\ \forall(\alpha) (\beta \to \beta) [\beta \leftarrow \alpha \to \alpha] \end{cases}
The definition of $i\text{ML}^F$

Types are stratified

$$\sigma ::= \begin{cases} \tau & \in F \\ \forall (\alpha \geq \sigma) \sigma \end{cases}$$

We can see and explain types by $\leq_F$-closed sets of System-F types:

$$\begin{align*}
\{\tau\} & \triangleq \{\tau' \mid \tau \leq_F \tau'\} \\
\{\forall (\alpha \geq \sigma) \sigma'\} & \triangleq \{\forall (\bar{\beta}) \tau'[\alpha \leftarrow \tau] \mid \land \left( \begin{array}{l} \tau \in \{\sigma\} \land \tau' \in \{\sigma'\} \\
\bar{\beta} \not\in \text{ftv}(\forall (\alpha \geq \sigma) \sigma') \end{array} \right)\}
\end{align*}$$

Type instance $\leq_I$ is set containment on the translations

$$\sigma \leq_I \sigma' \iff \{\sigma\} \supseteq \{\sigma'\}$$
Simple types

\[ \alpha \rightarrow \alpha \]
**Simple types**

\[ \alpha \rightarrow \alpha \]
Simple types

\[ \alpha \rightarrow \alpha \]
System-F types

\[ \forall (\alpha) \quad \alpha \rightarrow \alpha \]
System-F types

\[ \forall (\alpha) \quad \alpha \rightarrow \alpha \]
System-F types

\[ \forall (\alpha) \forall (\beta) (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \]
System-F types

\[ \forall \alpha \forall \beta (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \]
System-F types

\[ \forall(\alpha) \forall(\beta) (\alpha \to \beta) \to \alpha \to \beta \]
System-F types

\[(\alpha \to \beta) \to \alpha \to \beta\]

Sharing of inner nodes:
- Coming from the dag-representation of simple types.
- Canonical (unique) representation if disallowed.
System-F types

$$\forall (\alpha) \ (\forall (\beta) \ (\alpha \rightarrow \beta)) \rightarrow \alpha \rightarrow \beta$$

$$\forall (\alpha) \ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$
System-F types

\[ \forall (\alpha) \forall (\beta) (\alpha \to \beta) \to \alpha \to \beta \]

\[ \forall (\alpha) \forall (\gamma) (\alpha \to \gamma \to \gamma) \to \alpha \to \gamma \to \gamma \]
System-F types

\( \forall (\alpha) \forall (\beta) (\alpha \to \beta) \to \alpha \to \beta \)

\( \forall (\alpha)(\alpha \to \forall (\gamma) \gamma \to \gamma) \to \alpha \to \forall (\gamma) \gamma \to \gamma \)
Types in $i\text{MLF}$

$$\forall (\beta \geq \forall (\alpha) \; \alpha \rightarrow \alpha) \; \beta \rightarrow \beta$$
Types in $i\text{MLF}^F$

$$\forall (\beta \geq \forall (\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta$$
Types in $i\text{MLF}$

\[
\forall (\beta \geq \forall (\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta
\]
Types in $i\text{ML}^F$

$$\forall (\beta \geq \forall (\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta$$
Types in $\text{iML}^F$

$\forall (\beta \geq \forall (\alpha) \alpha \to \alpha) \beta \to \beta$
Types in $i\text{ML}^F$

\[ \forall (\beta \geq \forall (\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta \]

\[ (\forall (\alpha) \alpha \rightarrow \alpha) \rightarrow \forall (\alpha) \alpha \rightarrow \alpha \]
Types in $i\text{ML}_F^F$

\[ \forall (\beta \geq \forall (\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta \]

\[ \forall (\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \]
Types in $i\text{ML}^F$

$\forall (\beta \geq \forall (\alpha ) \alpha \to \alpha ) \beta \to \beta$
Types in $iML^F$

$$\forall (\beta \geq \forall (\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta$$
Types in $i\text{ML}^F$

$$\forall(\beta \geq \forall(\alpha)\alpha \rightarrow \alpha)\beta \rightarrow \beta$$
The semantics cannot be captured by
- a finite set of System-F types up to $\leq_F$
- a finite intersection type.
\( \forall (\beta \geq (\forall (\alpha) \, \alpha \rightarrow \alpha) \rightarrow (\forall (\alpha) \, \alpha \rightarrow \alpha)) \rightarrow \beta \rightarrow \beta \)
\( \forall (\beta \geq (\forall (\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall (\alpha) \alpha \rightarrow \alpha)) \rightarrow \beta \rightarrow \beta \)
\[ \forall (\beta \geq \left( \forall (\alpha) \alpha \to \alpha \right) \to \left( \forall (\alpha) \alpha \to \alpha \right)) \to \beta \to \beta \]
iMLF types

∀(β ≥ (∀(α) α → α) → (∀(α) α → α)) → β → β
Type instance $\leq$ in $i\text{ML}^F$

Only four atomic instance operations, and only two new.
Type instance $\leq$ in $i\text{ML}^F$

Only four atomic instance operations, and only two new.

Grafting

![Diagram showing grafting in $i\text{ML}^F$]

Didier Rémy (INRIA-Rocquencourt)
Type instance $\leq$ in $iML^F$

Only four atomic instance operations, and only two new.

Grafting

Raising
Type instance $\leq$ in $iML^F$

Only four atomic instance operations, and only two new.

Grafting

Raising

Merging
Type instance $\leqslant$ in $i\text{ML}^F$

Only four atomic instance operations, and only two new.

- **Grafting**
- **Raising**
- **Merging**
- **Weakening**
Type instance $\leq$ in $iML^F$

**Only four atomic instance operations, and only two new.**

- Grafting
- Raising
- Merging
- Weakening

- Merging only allowed on nodes transitively bound at the root (blue).
- Other operations only disallowed on variable nodes that are not transitively bound at the root (red).
Type instance $\leq$ in $iML^F$

Only four atomic instance operations, and only two new.

Grafting  \quad Raising  \quad Merging  \quad Weakening

These operations are sound and complete for the definition of $\leq$.

Can always be ordered as $\leq^G$; $\leq^R$; $\leq^{MW}$. 

Didier Rémy (INRIA-Rocquencourt)
A new look at ML$^F$
June 2008
Checking the example choice id

Raising

Weakening
Outline

1. Design
   - $i\text{ML}^F$: an implicitly-typed extension of System $F$
   - Types explained
   - $e\text{ML}^F$: an explicitly-typed version of $i\text{ML}^F$

2. Results
   - Principal types
   - Robustness to program transformations
   - Practice

3. Type inference
   - Type constraints for simple types
   - Type constraints for $\text{ML}$
   - Type inference in $\text{ML}^F$

4. Concluding remarks
Design of $eML^F$

**Goal**

Find a restriction $iML^F$ where programs that would require **guessing** polymorphism are ill-typed.

**Guideline**

Function parameters that are **used polymorphically** (and only those) need an annotation.
First-order inference with second-order types

Easy examples

\[ \lambda(z) \, z \quad : \quad \forall(\alpha) \, \alpha \to \alpha \quad \text{as in ML} \]

let \( x = \lambda(z) \, z \) in \( x \, x \) : \( \forall(\alpha) \, \alpha \to \alpha \) as in ML

\[ \lambda(x) \, x \, x \quad : \quad \text{ill-typed!} \quad x \text{ is used polymorphically} \]

\[ \lambda(x : \forall(\alpha) \, \alpha \to \alpha) \, x \, x \quad : \quad (\forall(\alpha) \, \alpha \to \alpha) \to (\forall(\alpha) \, \alpha \to \alpha) \]
First-order inference of second order types

More challenging examples

\((\lambda (z) z) \ (a : \sigma)\) where \(\sigma\) is truly polymorphic

- \(z\) carries values of a polymorphic type.
First-order inference of second order types

More challenging examples

\[(\lambda(z) z) \ (a : \sigma) \quad \text{where } \sigma \text{ is truly polymorphic} \]

- \(z\) carries values of a polymorphic type.
- but \(z\) is not used polymorphically.
- Indeed, it can be typed in System F as

\[(\Lambda \alpha. \ \lambda(z : \alpha) z ) \ [\sigma] \ (a : \sigma)\]
First-order inference of second order types

More challenging examples

\[ \lambda(z)\ (z\ (a : \sigma)) \]

- \( z \) must have a polymorphic type \( \sigma \rightarrow \tau \).
First-order inference of second order types

More challenging examples

\[ \lambda(z)(z \,(a : \sigma)) \]

- \( z \) must have a polymorphic type \( \sigma \rightarrow \tau \).
- \( z \) need not be used polymorphically: it may just carry polymorphism without using it.
- Indeed, it is the reduct of

\[ (\lambda(y) \, \lambda(z)(z \, y)) \quad (a : \sigma) \]

which can be typed in \( \text{MLF} \), exactly as the previous example.

Annotations need not be introduced during reduction!
Abstracting second-order polymorphism as first-order types

Solution

1) Disallow second-order types under arrows, e.g. such as $\sigma_{id} \rightarrow \sigma_{id}$
2) Instead, allow type variables to stand for polymorphic types:

   write $\forall (\alpha \ \sigma_{id}) \alpha \rightarrow \alpha$
   read "$\alpha \rightarrow \alpha$ where $\alpha$ abstracts $\sigma_{id}$"
   means $\sigma_{id} \rightarrow \sigma_{id}$

Mechanism

1) Function parameters must be monomorphic (but may be abstract).
2) Forces all polymorphism to be abstracted away in the context.

$$\alpha \Rightarrow \sigma_{id}, x : \alpha \vdash x : \alpha$$

$$\alpha \Rightarrow \sigma_{id} \vdash \lambda(x) x : \alpha \rightarrow \alpha$$

$$\lambda(x) x : \forall (\alpha \Rightarrow \sigma_{id}) \alpha \rightarrow \alpha$$
Abstracting second-order polymorphism

Key point: abstraction is directional

\[
\alpha \Rightarrow \sigma \vdash \sigma \leq \alpha
\]

Hence,

\[
\vdash a : \sigma
\]

\[
\alpha \Rightarrow \sigma \vdash a : \alpha
\]

\[
\alpha \Rightarrow \sigma \vdash \lambda(z) \, z \, a : (\alpha \rightarrow \alpha) \rightarrow \alpha
\]

\[
\vdash \lambda(z) \, z \, a : \forall (\alpha \Rightarrow \sigma) \, (\alpha \rightarrow \alpha) \rightarrow \alpha
\]
Abstracting second-order polymorphism

**Key point: abstraction is directional**

\[ \alpha \Rightarrow \sigma \vdash \sigma \leq \alpha \]

But,

\[ \alpha \Rightarrow \sigma_{id}, \ z : \alpha \vdash z : \alpha \]

\[ \alpha \Rightarrow \sigma_{id}, \ z : \alpha \vdash z : \sigma_{id} \]

\[ \alpha \Rightarrow \sigma_{id}, \ z : \alpha \vdash z : \alpha \rightarrow \alpha \]

\[ \alpha \Rightarrow \sigma_{id}, \ z : \alpha \vdash z : \alpha \rightarrow \alpha \]

\[ \alpha \Rightarrow \sigma_{id} \vdash \lambda(z) \ z \ z : \alpha \rightarrow \alpha \]

\[ \vdash \lambda(z) \ z \ z : \forall(\alpha \geq \sigma_{id}) \alpha \rightarrow \alpha \]
Types in eMLF

Introduce a new binder for abstraction

\[ \forall (\alpha \Rightarrow \forall (\beta) \, \beta \to \beta) \, \alpha \to \alpha \]
Types in eML$^F$

Introduce a new binder for abstraction

\[ \forall (\alpha \Rightarrow \forall (\beta) \beta \rightarrow \beta) \alpha \rightarrow \alpha \]
Types in eML$^F$

Introduce a new binder for abstraction

$$\forall (\alpha \Rightarrow \forall (\beta) \; \beta \rightarrow \beta) \; \forall (\alpha' \Rightarrow \forall (\beta) \; \beta \rightarrow \beta) \; \alpha \rightarrow \alpha'$$

More general sharing of $\Rightarrow$ matters
Types in eMLF

Introduce a new binder for abstraction

\[ \forall (\alpha \Rightarrow \forall (\beta) \beta \rightarrow \beta) \forall (\alpha' \geq \forall (\beta) \beta \rightarrow \beta) \alpha \rightarrow \alpha' \]

Even more general

\[ \geq \text{ better than } \Rightarrow \]
Types, graphically

= first-order term-dag + a binding tree
Types, graphically

$=$ first-order term-dag + a binding tree
Types, graphically

\( = \text{first-order term-dag} + \text{a binding tree} \)
Types, graphically

\[ = \text{first-order term-dag} + \text{a binding tree} \]

\[ + \text{well-formedness conditions relating the two} \]
Type instance $\preceq$ in $\text{eML}^F$

Sharing and binding of abstract nodes now matter

Grafting, Merging, Raising, Weakening
Unchanged.
Type annotations

Recovering the missing power

\[(\leq) \subset (\leq_I)\]

\(\leq\) is weaker than \(\leq_I\), as sharing and binding of abstract nodes matters.
Type annotations

Recovering the missing power

\[(\preceq) \subseteq (\preceq_1) = (\preceq \cup \preceq_1)^*\]

- \(\preceq\) is weaker than \(\preceq_1\), as sharing and binding of abstract nodes matters.
- Use explicit type annotations to recover \(\preceq_1 \setminus \preceq\).

Notice that the larger \(\preceq\), the fewer type annotations.
Type annotations

Recovering the missing power

\[(\leq) \subset (\leq_I) = (\leq \cup \leq_I)^*\]

Technically

- Intuitively,

\[
\Gamma \vdash a : \tau \quad \tau \leq_I \tau' \\
\hline
\Gamma \vdash (a : \tau') : \tau'
\]

- Actually, use coercion functions:

\[
(\_ : \sigma) : \quad \forall (\alpha \Rightarrow \sigma) \forall (\alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha'
\]

- Add syntactic sugar \(\lambda(x : \sigma) \ a \triangleq \lambda(x) \ \text{let} \ x = (x : \sigma) \ \text{in} \ a\)

\[
\equiv \lambda(x) \ a[x \leftarrow (x : \sigma)]
\]
Type annotations

Recovering the missing power

\[(\leq) \subset (\leq_I) = (\leq \cup \triangleleft_I)^*\]

Technically

- Intuitively,

\[
\Gamma \vdash a : \tau \quad \tau \triangleleft_I \tau' \\
\hline
\Gamma \vdash (a : \tau') : \tau'
\]

- Actually, use coercion functions:

\[
(_ : \exists(\bar{\beta}) \sigma) : \forall(\bar{\beta}) \forall(\alpha \Rightarrow \sigma) \forall(\alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha'
\]

- Add syntactic sugar \(\lambda(x : \sigma) a\)

\[\triangleq \lambda(x) \text{ let } x = (x : \sigma) \text{ in } a\]

\[\equiv \lambda(x) a[x \leftarrow (x : \sigma)]\]
Type annotations

Remember \( \alpha \Rightarrow \sigma, x : \alpha \vdash x : \sigma \)

- Prevents typing \( \lambda(x) \ x \ x \)

With an annotation \( \alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \sigma \)

- Allows typing \( \lambda(x : \sigma_{id}) \ x \ x \)
Outline

1 Design
- $i\text{ML}^F$: an implicitly-typed extension of System F
- Types explained
- $e\text{ML}^F$: an explicitly-typed version of $i\text{ML}^F$

2 Results
- Principal types
- Robustness to program transformations
- Practice

3 Type inference
- Type constraints for simple types
- Type constraints for ML
- Type inference in $\text{ML}^F$

4 Concluding remarks
Principal types

Fact
- Programs have principal types, given with their type annotations.

Programs with type annotations
- Two versions of the same program, but with different type annotations, usually have different principal types.

Programs typable without type annotations
- Exactly ML programs.
- But usually have a more general type than in ML (e.g. choice id)
- Annotations may still be useful to get more polymorphism.
Robustness to program transformations

Agreed

- Programmers must be free of choosing their programming patterns/styles.
- Programs should be maintainable.

Therefore

- Programs should be stable under some small, but important program transformations.
Robustness to program transformations

\( a \subseteq a' \) means all typings of \( a \) are typings of \( a' \)

**Let-conversion**

\[(x \in a_2) \; \text{let} \; x = a_1 \; \text{in} \; a_2 \supseteq a_2[x \leftarrow a_1] \]

Common subexpression can be factored out.

**\( \eta \)-conversion of functional expressions**

\[a \supseteq \lambda(x) a x\]

Delay the evaluation.

**Redefinable application**

\[a_1 a_2 \supseteq (\lambda(f) \lambda(x) f x) a_1 a_2\]

Many functionals, such as maps are typed as applications.

**Reordering of arguments**

\[a a_1 a_2 \supseteq (\lambda(x) \lambda(y) a y x) a_2 a_1\]

**Curryfication**

\[a (a_1, a_2) \supseteq (\lambda(x) \lambda(y) a (x, y)) a_1 a_2\]

All valid in ML\(^F\)
Robustness to program transformations

Reduction

- Transforms existing type annotations
- Does not introduce new type annotations
Printing types

Problem

- Types are graphs.
- They can be represented syntactically with prefix notation,
- Not very readable: compare \((\forall(\gamma)\gamma \to \gamma) \to (\forall(\gamma)\gamma \to \gamma)\)
  with \(\forall(\alpha \Rightarrow \forall(\gamma)\gamma \to \gamma)\forall(\beta \geq \forall(\gamma)\gamma \to \gamma)\alpha \to \beta\)

Solution

- Inline linear bindings that are
  - flexible at covariant positions, or
  - rigid at contravariant positions.

- Very effective in practice: types look often as in System F.
Examples

Library functions

```ml
let rec fold f v = function
  | Nil  -> v
  | Cons (h, t) -> fold f (f h t) t

val fold : \forall (\alpha) \forall (\beta) (\alpha \rightarrow \alpha list \rightarrow \beta) \rightarrow \beta \rightarrow \alpha list \rightarrow \beta
```

Few type annotations are needed in practice

- No dummy/annoying/unpredictable annotations.

Output types are usually readable

- Most inner binding edges may be left implicit.
- Many library functions libraries keep their ML type in ML\(^F\), modulo the syntactic sugar.
Outline

1 Design
   - $i\text{ML}_F$: an implicitly-typed extension of System F
   - Types explained
   - $e\text{ML}_F$: an explicitly-typed version of $i\text{ML}_F$

2 Results
   - Principal types
   - Robustness to program transformations
   - Practice

3 Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in $\text{ML}_F$

4 Concluding remarks

Didier Rémy (INRIA-Rocquencourt)

A new look at $\text{ML}_F$

June 2008
Type inference for ML

Based on first-order unification

- Best implemented and formalized using graphs (Huet) or, equivalently, multi-equations.

Type inference

- Best formalized by type constrains
- See *The essence of ML, in ATTAPL.*
First-order unification

Unification problems may be represented on a term using unification edges
First-order unification

Unification problems may be represented on a term using unification edges
First-order unification

Unification problems may be represented on a term using unification edges

Merging
First-order unification

Unification problems may be represented on a term using unification edges

Grafting
First-order unification

Unification problems may be represented on a term using unification edges

Merging
First-order unification

Unification problems may be represented on a term using unification edges

Unification builds a dag, by merging variables or inner nodes,
First-order unification

Unification problems may be represented on a term using unification edges

- Unification builds a dag, by merging variables or inner nodes,
- However, the dag may be read back up to sharing of inner nodes.
  - Because extra sharing will never block further simplifications
  - Thus, inner nodes could always be maximally shared (hash-consing).
  - Hence, sharing of inner nodes does not matter.
Unification formally

Term view
A solution to a unification problem is an instance in which subterms connected by unification edges are equal.

Graph view (simpler)
A solution to a unification problem is an instance in which nodes connected by unification edges are identical.

The algorithm (with simple proof of correctness)
- Each transformation preserves the set of solutions.
- Applying transformations terminates, with either:
  - an obvious conflict, thus, there is no solution; or
  - a graph without constraints, hence a solution of which all others are instances, i.e. a principal solution.
Type inference for simply typed $\lambda$-calculus

It is well-known that it reduces to unification problems:
Type inference for simply typed \( \lambda \)-calculus

Example:

\[
(\lambda(f) \lambda(x) f \, x) \, (\lambda(y) y)
\]

Graphically: the \( \lambda \)-term
Type inference for simply typed $\lambda$-calculus

Example:

\[
\left( \lambda(f) \lambda(x) f \, x \right) \left( \lambda(y) y \right)
\]

Graphically: its type constraint
Type inference for simply typed $\lambda$-calculus

Example:

$$\left( \lambda(f) \lambda(x) f \, x \right) \left( \lambda(y) y \right)$$

Graphically: its type constraint
Type inference for simply typed $\lambda$-calculus

Example:

\[
\left( \lambda(f) \lambda(x) f \ x \right) \ (\lambda(y) \ y)
\]

Graphically: its type constraint
Type inference for simply typed $\lambda$-calculus

Example:

$$(\lambda(f)\,\lambda(x)\, f\, x)\, (\lambda(y)\, y)$$

Graphically: its type constraint
Type inference for simply typed $\lambda$-calculus

Example:

$$(\lambda(f) \lambda(x) f \ x) \ (\lambda(y) y)$$

Graphically: its type constraint
Type inference for simply typed $\lambda$-calculus

Example:

$$(\lambda(f) \lambda(x) f \, x) \, (\lambda(y) \, y)$$

Graphically: its type constraint
Type inference for simply typed $\lambda$-calculus

Example:

$$\left( \lambda(f) \lambda(x) f \ x \right) \left( \lambda(y) y \right)$$

Graphically: its solved form
Constraint generation

Variables

Functions

Applications
Constraint generation

Variables

Functions

Applications

Didier Rémy (INRIA-Rocquencourt)  A new look at MLF  June 2008  (2) 38 / 57
Constraint generation

Variables

Functions

Applications

Bindings
Type inference for let-bindings

Can we extend the previous schema?

- The question is usually eluded in books.
- The solution is type inference with let-constraints.
- Can be better explained graphically.
Type inference for let-bindings

Introduce G-nodes (Generalization points) to represent type schemes and distinguish them from types

\[ \forall (\alpha \beta) \, (\alpha \to \beta) \to \gamma \]

Generalized variables are drawn as binding edges to G.
Type inference for let-bindings

**Introduce G-nodes** (Generalization points) to represent type schemes and distinguish them from types

Generalized variables are drawn as binding edges to G. Inner nodes may also be bound to G-nodes.
**Type inference for let-bindings**

*Introduce G-nodes* (Generalization points) to represent type schemes and distinguish them from types.

\[
\forall (\alpha \beta) \ (\alpha \to \beta) \to \gamma
\]

Generalized variables are drawn as binding edges to G.

**Constraint generation**

Expressions now represent G-nodes., i.e. type scheme constraints.
Constraint generation for ML (revisited)

Variables

- Variables are represented by a square labeled with "G".

- Functions are represented by a lambda symbol "\( \lambda \)".

- Applications are represented by a circle labeled with "@".

- Variables are mapped to a type constraint "\( 1 \rightarrow G \)".

- Functions are mapped to a type constraint "\( \cdot \rightarrow 1 \rightarrow G \)".

- Applications are mapped to a type constraint "\( 1 \rightarrow 2 \rightarrow G \)".
Constraint generation for ML (revisited)

Let-bindings

\[ \text{let} \Rightarrow \]

\[ \lambda \text{-bindings} \]

\[ \text{let-bindings} \]

\[ \Rightarrow \]
Example:

\[
\text{let } g = \lambda(x) \cdot x \text{ in } g \ (\lambda(y) \cdot y)
\]

Graphically (on the right):

the $\lambda$-term
Example:

\[ \text{let } g = \lambda(x)\, x \text{ in } \] 
\[ g\ (\lambda(y)\, y) \]

Graphically (on the right):

its type constraint
Example:

\[
\text{let } g = \lambda(x) \ x \ \text{in} \\
g \ (\lambda(y) \ y)
\]

Graphically (on the right):

its type constraint

Superfluous generalization points

- As in ML generalization is only needed at let-bindings.
- Useless G-nodes may be simplified after/during constraint generation.
Well-formedness of constraints

Well-formedness

- Arities: all nodes have a fixed number of outgoing structure edges
- Kinds: we distinguish G-nodes from other, regular nodes.
  - Instantiation edges are from G-nodes to regular nodes.
  - Unification edges are between from regular nodes.
- All nodes are bound to some G-node.
- The binding of a node is one of its dominators for mixed structure and binding edges.

Existential nodes

- Nodes that do not have an incoming structure edge.

Projection

- Remove all existential nodes and constraints.
A new instance operation

Raising a binding edge along another one. This amounts to treating a polymorphic as locally monomorphic.
A new instance operation

Raising a binding edge along another one.
This amounts to treating a polymorphic as locally monomorphic
A new instance operation

Raising a binding edge along another one.
This amounts to treating a polymorphic as locally monomorphic.
Solved instantiation edge

**Informally**
An instantiation edge is solved if its target is an instance of its origin.

**Expansion**
Solved instantiation edge

Informally
An instantiation edge is solved if its target is an instance of its origin.

Expansion

Expansion does not copy constraint edges nor existential nodes.
Solved instantiation edge

Informally
An instantiation edge is solved if its target is an instance of its origin.

Expansion

Can we come back to the original term by instantiation? —No
Solved instantiation edge

Informally
An instantiation edge is solved if its target is an instance of its origin.

Expansion

Can we come back to the original term by instantiation? —Yes
Solved instantiation edge

Informally
An instantiation edge is solved if its target is an instance of its origin.

Expansion
Expansion can be used as a test
Solved instantiation edge

Informally
An instantiation edge is solved if its target is an instance of its origin.

Expansion

Expansion can also be used as a simplification: the instantiation edge can be removed, if the origin is solved type scheme were solved.
Semantics of constraints

The set of its instances in which all constrained edges are solved.

**Constraint simplifications** (preserve the semantics)

- Solving a unification edge by unification (as before).
- Expansion-elimination of an instantiation edge whose origin is solved.
- Garbage collection of unconstrained existential nodes.
- Elimination of superfluous G-nodes.
Algorithm

1. Eliminate superfluous G-nodes first, for efficiency.
2. Solve unification edges eagerly.
3. Solve instantiation constraints, innermost first.
4. Garbage collect at any time (no efficiency impact).

Complexity in $O(kn(\alpha(kn) + d)) \approx O(kdn)$ (see McAllester)

- $k$ is the maximal size of types (usually not too large)
- $d$ is the maximal nesting of type schemes
  i.e. after simplification of useless generalizations, let-nesting of
  let-bindings (reasonably below 5).

Explains why ML type inference works well in practice

- Large programs mainly increase right nesting of let-bindings.
**Unification algorithm**

**Computes principal unifiers, in three steps**
- Computes the underlying dag-structure by first-order unification.
- Computes the binding structure
  - by raising binding edges
  - as little as possible to maintain well-formedness.
- Checks that no locked binding edge (in red) has been raised or merged.

**Complexity**
- Same as first-order unification. Other passes are in linear time.
- \( O(n) \) (or \( O(n\alpha(n)) \) if incremental).

**Note**
- The algorithm performs "first-order unification of second-order types".
Type inference

Proceeds much as in ML, except that

- Generalize as much as possible at every step (not just at every let).
- Nodes may be bound to G-nodes or other nodes.
- Existential nodes only bound to G-nodes.
- Expansion is modified to reset topmost bindings:

In particular, constraint generation is unchanged.
Type inference

Complexity, also in $O(kn(\alpha(kn) + d)) \approx O(kdn)$
However, ML and ML$^F$ differs on $d$, which is:

- the left-nesting of let-bindings in ML
- the maximum height of an expression in ML$^F$
  (Still, does not grow on the right of let-bindings).
Outline

1 Design
   - $i\text{ML}^F$: an implicity-typed extension of System F
   - Types explained
   - $e\text{ML}^F$: an explicitly-typed version of $i\text{ML}^F$

2 Results
   - Principal types
   - Robustness to program transformations
   - Practice

3 Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in $\text{ML}^F$

4 Concluding remarks
Variations on $\text{ML}^F$

**Shallow $\text{ML}^F$**

The version we presented is a “downgraded” version of $\text{ML}^F$.

- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be $F$ types.
Variations on $\text{ML}^F$

**Shallow $\text{ML}^F$**
The version we presented is a “downgraded” version of $\text{ML}^F$.

- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be F types.

**Full $\text{ML}^F$**

- No stratification, more expressive.
- All interesting properties are preserved.
- Algorithms are mostly unchanged.
- We lose the interpretation of types as sets of System-F types.
Variations on $\text{MLF}$

**Shallow $\text{MLF}$**
The version we presented is a “downgraded” version of $\text{MLF}$.
- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be F types.

**Simple $\text{MLF}$**
Remove instance bindings $\geq$, keep abstract bindings $\Rightarrow$.
- Equivalent to System F.
- Principal types are lost (no type inference).
Variations on ML$^F$

**Shallow ML$^F$**
The version we presented is a “downgraded” version of ML$^F$.
- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be F types.

**Simple ML$^F$**
Remove instance bindings $\geq$, keep abstract bindings $\Rightarrow$.
- Equivalent to System F.
- Principal types are lost (no type inference).

**Is there an interesting variant in between?**
- As expressive as System F.
- With type inference and principal types. **Yes! Leijen’s HML**
A hierarchy of languages

(Full) ML$^F$  \rightarrow  Shallow ML$^F$

\begin{align*}
F & \not\geq \Rightarrow \\
ML & \geq \Rightarrow \not\Rightarrow \\
Simple Types & \Rightarrow \n
\end{align*}
A hierarchy of languages

(Full) ML\textsuperscript{F} ➔ λ-∀≥

Shallow ML\textsuperscript{F} ➔ let-∀≥

F ➔ λ-∀

ML ➔ let-∀

Simple Types ➔ Simple ML\textsuperscript{F} ➔ let-∀

Simple ML\textsuperscript{F} ➔ Shallow ML\textsuperscript{F} ➔ λ-∀≥

Simple Types ➔ F ➔ λ-∀
A hierarchy of languages

(Full) ML^F

Shallow ML^F

HML

Simple ML^F

ML

Simple Types

F

Simple Types
A hierarchy of languages
An internal language for $\text{ML}^F$ (on going work)

**Problem**

- $\text{iML}^F$ is in curry style.
- $\text{eML}^F$ is not quite in church style:
  - type reconstruction is non local
  - type annotations must be transformed during reduction,
    but $\text{eML}^F$ does not describe how to do so.
- Need for a church-style $\text{ML}^F$ (*e.g.* compiling Haskell)

**Solution**

- Make type abstraction and type application fully explicit,
- Annotate all parameters of functions,
- Use a more general form of type application that witness the correct type-instantiation.
Extensions

Primitive Existential types

- Encoding with existential types works well (only annotate at creation).
- Can more be done with primitive existential?

(Equi-) recursive types

- Easy when cycles do not contain quantifiers.
- Cycles that cross quantifiers are difficult.

Higher-order types

- Use two quantifiers (explicit coercions between the two permitted)
  - $\forall^F$ for fully explicit type abstractions and
  - $\forall^{MLF}$ for implicit MLF polymorphism.
- Restrict $\forall^{MLF}$ to the first-order type variables.
- Can $\forall^{MLF}$ also be used at higher-order kinds?
Conclusions

To bring back home

- $\text{ML}^F$ allows function parameters to implicitly carry polymorphic values that are used \textit{monomorphically}.
- Type annotations are required only to allow function parameters to carry (polymorphic) values that are used \textit{polymorphically}.

$\text{ML}^F$ design, use, and implementation are close to ML

- $\text{ML}^F$ piggy-backs on ML type-schemes and generalization mechanism.
- Part of the credits should be returned to the great designer of ML.

Hopefully

- ML users will feel “\textit{at home}”.
- Other users will also appreciate the convenience of type inference.
Papers and prototypes

Talk mainly based on

- Recasting-$\text{ML}^F$ with Didier Le Boltan.
- Graphic Type Constraints and Efficient Type Inference: from ML to $\text{ML}^F$, with Boris Yakobowski.

Other papers and online prototype at

- http://gallium.inria.fr/~remy/mlf/

See also Daan Leijen’s papers and prototypes ($\text{HMF}$, $\text{HML}$)

- http://research.microsoft.com/users/daan/pubs.html

and works by Vytinioitis et al. (Boxy types, FPH)

- http://research.microsoft.com/users/daan/pubs.html
Appendix

5 Printing types

6 More examples
   - Church numerals
   - encoding of existential types

7 Other restrictions of $\text{ML}^F$

8 Questions
   - Sharing of abstract nodes is irreversible (implicitly)
   - Stability by linear beta-expansion

9 Details of slides
   - Another example of System $\text{F}$ types
   - Abstraction in action

10 Type inference demo
Printing types

Only overlined bindings need to be drawn

Leave implicit bindings that are

- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.

\[(\forall (\alpha) \forall (\beta) (\alpha \to \beta) \to (\alpha \to \beta)) \to (\forall (\alpha) \alpha \to \alpha) \to (\forall (\alpha) \alpha \to \alpha)\]
Printing types

Only overlined bindings need to be drawn

Leave implicit bindings that are

- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.

\[(\forall (\alpha) \forall (\beta) (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \rightarrow \forall (\gamma \Rightarrow \forall (\alpha) \alpha \rightarrow \alpha) \left( (\forall (\alpha) \alpha \rightarrow \alpha) \rightarrow \gamma \right) \]
Only overlined bindings need to be drawn

Leave implicit bindings that are

- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.

\[
\forall (\gamma \Rightarrow \forall (\alpha) \alpha \rightarrow \alpha) \left( \forall (\alpha) \forall (\beta) (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \right) \rightarrow (\forall (\alpha) \alpha \rightarrow \alpha) \rightarrow \gamma
\]
Printing types

Only overlined bindings need to be drawn

\[(\forall \alpha) \ (\forall \beta) \ (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)\] \rightarrow \forall (\gamma \geq \sigma_{id}) \ \gamma \rightarrow \gamma

Leave implicit bindings that are

- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.
back
More examples

Church numerals

type nat = ∀ (α) (α → α) → α → α;;

let zero = fun f x → x;;
val zero : ∀(α) α → (∀(β) β → β)

With type annotations on the iterator

let succ (n : nat) = fun f x → n f (f x);;
val succ : nat → (∀ (α) (α → α) → α → α)

let add (n : nat) m = n succ m;;
val add : nat → (∀ (α) (α → α) → α → α)

let mul n (m : nat) = m (add n) zero;;
mul : nat → nat → (∀(α) (α → α) → α → α)
More examples

Church numerals

type nat = ∀ (α) (α → α) → α → α;;

let zero = fun f x → x;;
val zero : ∀(α) α → (∀(β) β → β)

Without type annotations

let succ n = fun f x → n f (f x);;
val succ : ∀ (α, β, γ) ((α → β) → β → γ) → (α → β) → α → γ

let add n m = n succ m;;
val add : ∀(δ ≥ ∀(α, β, γ) ((α → β) → β → γ) → (α → β) → α → γ)
∀(ε, ϕ) (δ → ε → ϕ) → ε → ϕ

In ML:

val add : ∀ (α, β, γ, ε, ϕ) (((((α → β) → β → γ) → (α → β) → α → γ)
→ ε → ϕ) → ε → ϕ
More examples

Church numerals

type nat = ∀ (α) (α → α) → α → α;;
let zero = fun f x → x;;
val zero : ∀(α) α → (∀(β) β → β)

Mandatory type annotations

let succ n = fun f x → n f (f x);;
let succ' = (succ : nat → nat);;
fails

ML^F without any type annotation at all does not do better than ML!
More examples

Encoding of existential types, e.g. \( \exists \beta. \beta \times \beta \to \alpha \)

type \( \alpha \) func = \(\forall (\gamma) \forall (\delta = \forall (\beta) \beta \times (\beta \to \alpha) \to \gamma) \) \( \delta \to \gamma \)

val pack \( z = \) fun \( f : \exists (\gamma) \forall (\beta) \beta \times (\beta \to \alpha) \to \gamma \) \( \to f \ z ;; \)

val pack : \( \forall (\alpha) \forall (\beta) \alpha \times (\alpha \to \beta) \to (\forall (\gamma) (\forall (\delta) \delta \times (\delta \to \beta) \to \gamma) \to \gamma) \)

let packed_int = pack (1, fun \( x \to x + 1) ;; \)

let packed_pair = pack (1, fun \( x \to (x, x) ;; \)

let \( v = \) packed_int (fun \( p \to (snd \ p) (fst \ p) ) ;; \)
HML: no rigid bindings

HML, proposed by Daan Leijen

- the specification uses the same types as $i\text{ML}^F$.

A strict subset of $\text{ML}^F$

- annotate exactly arguments that are used polymorphically.
- can be explained as follows;
  - Disable rigid bindings in prefixes.
  - Then, abstraction commutes with type inference.
  - Hence, types may be treated up to abstraction. bindings.

Gains and losses

- Simpler, more intuitive types.
- Keep most essential properties (principal types, robustness)
- Lost of some robustness. Polymorphism is not quite first-class.
  e.g., primitive integers can’t be replaced by church numerals.
FPH: only System-F like types in the specification

HML can be further restricted

- The specification uses only System-F types.

Many losses

- Inference algorithm is kept (using $\text{ML}^F$ internally...)
- Bigger lost of some robustness.
- No longer principal types per se.

Two variants to recover principal derivations

- HML: imposes minimal rank of polymorphism when ambiguous, which may require type annotations to get deeper polymorphism.
- FPH: requires no ambiguity at let-bindings, which may require type annotations to disambiguate.
Rigid ML$^F$

Rigid ML$^F$ lies very close to ML$^F$

- It uses and relies on (Shallow) ML$^F$ internally.
- It projects ML$^F$ principal types into System-F types at let-bindings, by raising variable bindings as much as possible.

Rigid ML$^F$ looses important properties of ML$^F$

- There are no principal types *per se*.
  - Rigid ML$^F$ pretends to have principal types, but this is in an ad hoc manner, using a non logical typing rule for Let-bindings with a premise that blocks free uses of type-instantiation.

- let $x = \lambda(z : \sigma) z$ in $a_2$ may be accepted while let $x = \lambda(z) z$ in $a_2$ would be rejected.

- Rigid ML$^F$ is not invariant by let-expansion (which signs the lost of truly principal types).
Rigid ML$^F$

Rigid ML$^F$ lies very close to ML$^F$

- It uses and relies on (Shallow) ML$^F$ internally.
- It projects ML$^F$ principal types into System-F types at let-bindings, by raising variable bindings as much as possible.

Rigid ML$^F$ looses important properties of ML$^F$

- There are no principal types *per se*.
- Rigid ML$^F$ is not invariant by let-expansion (which signs the lost of truly principal types).

Rigid ML$^F$ is a subset of System F

- This is both its interest and its problem.
Sharing of abstract nodes is irreversible (implicitly)
Can you show an example illustrating the difference?

**Fact:** \( \forall (\alpha \Rightarrow \sigma) \alpha \rightarrow \alpha \nless \forall (\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha' \)

Observe that:

- \( \lambda(z) z : \forall (\alpha \Rightarrow \sigma) \alpha \rightarrow \alpha \)
- \( (_\sigma) : \forall (\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha' \)

Then, the context \( a \triangleq \lambda(x) [] x x \) distinguishes those two expressions.

- \( a[\lambda(z) z] \) is ill-typed.
  (As it uses no type annotation and it is ill-typed in ML)
- \( a[(_\sigma)] \) is well-typed.
Stability by linear beta-expansion

Linear $\beta$-conversion?

No! otherwise, for $x \in a_1$:

\[
(\lambda (x) \ a_1) \ a_2 \supset\!
(\lambda^1 (x) \text{ let } x = x \text{ in } a_1) \ a_2 \\
\supset\!
(\text{let } x = x \text{ in } a_1)[x \leftarrow a_2] \supset\!
(\text{let } x = a_2 \text{ in } a_1)
\]

Linearity is misleading:

\[
\lambda^1 (x) \text{ let } y = x \text{ in } y \ y
\]

is not typable! Indeed, $x$ must be used polymorphically via $y$. 
back
System-F types (encoding of existential types)

\[ \forall (\alpha) \ (\forall (\beta) \ \tau_{\beta} \rightarrow \alpha) \rightarrow \alpha \]
System-F types (encoding of existential types)

\[ \forall(\alpha) (\forall(\beta) \tau_\beta \rightarrow \alpha) \rightarrow \alpha \]

\[ (\forall(\beta) \tau_\beta \rightarrow \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \forall(\alpha) \alpha \rightarrow \alpha \]
System-F types (encoding of existential types)

$$\forall (\alpha) \left( \forall (\beta) \; \tau_\beta \rightarrow \alpha \right) \rightarrow \alpha$$
Type annotations

\[
\alpha \Rightarrow \sigma, \beta \Rightarrow \sigma \vdash \sigma \leq \alpha \text{ and } \sigma \leq \beta
\]

\[
\alpha \Rightarrow \sigma, \beta \Rightarrow \sigma \vdash \forall (\alpha' \Rightarrow \sigma) \forall (\beta' \Rightarrow \sigma) \alpha' \rightarrow \beta'
\]

\[
\leq \forall (\alpha' \Rightarrow \alpha) \forall (\beta' \Rightarrow \beta) \alpha' \rightarrow \beta'
\]

\[
\triangleleft
\]

\[
\alpha \rightarrow \beta
\]

\[
\alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash (\_: \sigma) : \alpha \rightarrow \beta
\]

\[
\alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash \_ : \sigma
\]

\[
\alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash (x : \sigma) : \beta
\]

\[
\alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \forall (\beta \Rightarrow \sigma) \beta
\]

\[
\alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \sigma
\]
Type annotations

\[ \alpha \Rightarrow \sigma_{id}, x : \alpha \vdash (x : \sigma_{id}) : \sigma_{id} \]

\[ \alpha \Rightarrow \sigma_{id}, x : \alpha \vdash (x : \sigma_{id}) : \alpha \rightarrow \alpha \]

\[ \alpha \Rightarrow \sigma_{id}, x : \alpha \vdash x : \alpha \]

\[ \alpha \Rightarrow \sigma_{id} \vdash \lambda(x) (x : \sigma_{id}) x : \alpha \rightarrow \alpha \]

\[ \vdash \lambda(x) (x : \sigma_{id}) x : \forall (\alpha \Rightarrow \sigma_{id}) \alpha \rightarrow \alpha \]
Type inference with typing constraints (demo)

\[ \lambda(x) \ x \]
Type inference with typing constraints (demo)

\[ \lambda(x) \ x \]
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)

\[
\text{let } y = \lambda(x) \ x \\
\text{in } y \ y
\]
Type inference with typing constraints (demo)

let \( y = \lambda(x) \ x \)
in \( y \ y \)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)

\( \lambda(z) z (\lambda(x) x) \)
Type inference with typing constraints (demo)

\[ \lambda(z) \ z \ (\lambda(x) \ x) \]
Type inference with typing constraints (demo)

\[ \lambda(x) \, x \]
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)

\[ \lambda(z) \ (z : \sigma_{\text{id}}) \]
Type inference with typing constraints (demo)

\[ \lambda(z) \; (z : \sigma_{id}) \]
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)
Type inference with typing constraints (demo)