From ML to ML$^F$

Graphic type constraints with efficient type inference

Who? Boris Yakobowski, Didier Rémy

Where? INRIA, Gallium team

When? ICFP 2008
MLF

Extends both ML and System F, combining the benefits of both

**Compared to ML**
- The expressivity of first-class polymorphism is available
- All ML programs remain typable unchanged

**Compared to System F**
- MLF has type inference
- Programs have principal types (taking type annotations into account)

**Moreover:**
- In practice, programs require very few type annotations
- Typable programs remain typable under all expected program transformations
(Lack of) modularity of System F

System F does not have principal types

Programs cannot be typed modularly

Example

choose : $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$

id : $\forall \beta. \beta \rightarrow \beta$

choose id : \[
\left\{\begin{array}{l}
\forall \gamma. (\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma) \\
(\forall \beta. \beta \rightarrow \beta) \rightarrow (\forall \beta. \beta \rightarrow \beta)
\end{array}\right.
\]

No most general type in System F
Bounded quantification

**MLF types**

MLF types extend System F types with instance-bounded quantification $\forall (\alpha \geq \tau) \tau'$:

- All occurrences of $\alpha$ in $\tau'$ have a (same) instance of $\tau$
- Both $\tau$ and $\tau'$ can be instantiated

choose id : $\forall (\alpha \geq \forall \beta. \beta \rightarrow \beta) \alpha \rightarrow \alpha$

\[\subseteq (\forall \beta. \beta \rightarrow \beta) \rightarrow (\forall \beta. \beta \rightarrow \beta)\]

- taking $\alpha = \forall \beta. \beta \rightarrow \beta$

\[\subseteq \forall \gamma. (\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)\]

- taking $\alpha = \gamma \rightarrow \gamma$ for a fresh $\gamma$
Graphic types

An alternative representation of MLF types (or ML ones)

Simplify the meta-theory of MLF

A graphic type
The superposition of
a term-dag, representing the skeleton of the type

\[ \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \]

\[ \forall \alpha. \forall (\beta \geq \forall \gamma. \gamma \rightarrow \alpha) \beta \rightarrow \beta \]
Graphic types

An alternative representation of ML$^F$ types (or ML ones)
Simplify the meta-theory of ML$^F$

A graphic type

The superposition of

- a term-dag, representing the skeleton of the type
- a binding tree, indicating where and how variables are bound

∀$\alpha$. $(\alpha \to \alpha) \to (\alpha \to \alpha)$
∀$\alpha$. ∀$(\beta \geq \forall \gamma. \gamma \to \alpha) \beta \to \beta$
Graphic constraints

Used to perform ML or ML$^F$ type inference on graphic types
Graphic constraints

Used to perform \texttt{ML} or \texttt{MLF} type inference on graphic types

An \textit{extension} of graphic types (only three new constructs):

- unification edges
- generalization scopes
- instantiation edges

Very small extension: we can reuse all the existing results on graphic types
Graphic constraints

- Used to perform $\text{ML}$ or $\text{ML}^F$ type inference on graphic types

- An extension of graphic types (only three new constructs):
  - unification edges
  - generalization scopes
  - instantiation edges

  Very small extension: we can reuse all the existing results on graphic types

- Using constraints is more general than a type inference algorithm
  e.g. different solving strategies
Typing abstractions or applications graphically

\[
T(a \ b) = \exists \alpha, \exists \beta, \\
(\alpha \rightarrow \beta = T(a) \land \alpha = T(b)). \beta
\]

\[
T(\lambda(x) \ a) = \exists \alpha, \exists \beta, \\
(\alpha = T(x) \land \beta = T(a)). \alpha \rightarrow \beta
\]

Green arcs are unification edges

Circled nodes are the result type
Type generalization

Type generalization is needed in ML (and in ML^F)

We introduce special G-nodes in graphs to promote types to type schemes

\[ g : \forall \beta. \beta \rightarrow \beta \]
\[ g' : \forall \alpha. \alpha \rightarrow \beta \]

\( \beta \) is free at the level of \( g' \)

G-nodes are also used to delimit generalization scopes
(also, strong correspondance with ranks in efficient ML type inference)
Instantiation edge

Constrain a node to be an instance of a type scheme

\[ g' : \forall \alpha. \alpha \rightarrow \beta \]
\[ n : \beta \rightarrow \beta \]

\text{e is solved} \quad (\text{take } \alpha = \beta)
Instantiation edge

Constrain a node to be an instance of a type scheme

$$g : G \rightarrow \alpha \perp \rightarrow \perp$$

$$G'$$

$$e$$

$$g' : \alpha \rightarrow \beta$$

$$n : \beta \rightarrow \beta$$

$$e$$ is not solved \((\alpha \neq \beta)\)
Typing constraints

Source language: (MLF only)

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

Source language: (MLF only)

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

\(\lambda\)-terms are translated into typing constraints compositionnally

\[ a \] represents the typing constraint for \( a \)

The blue arrows are constraint edges (unification or instantiation) for the free variables of \( a \)
Typing constraints

Source language: \[(\text{MLF only})\]

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

\(\lambda\)-terms are translated into typing constraints compositionnally

One generalization scope by subexpression

in ML, only needed for let; in MLF, needed everywhere

Exact same typing constraints for ML and MLF

- the useless G-nodes can be removed in ML
- MLF constraints allow the more general types of MLF, and have a more general notion of generalization
Typing constraint for an application

\[ T(a \ b) = \text{GEN}(\exists \alpha, \exists \beta, (T(a) \sqsubseteq \alpha \rightarrow \beta \wedge T(b) \sqsubseteq \alpha). \beta) \]
Typing constraint for an abstraction

\[ \lambda(x) \ a \sim \rightarrow \]

\[ \exists \alpha, \exists \beta, (T(x) = \alpha \land T(a) \subseteq \beta). \alpha \rightarrow \beta \]
Typing constraint for a let

Each occurrence of $x$ in $b$ must have a (possibly different) instance of $\mathcal{T}(a)$
Typing constraint for variables

A trivial type scheme \((\forall \alpha. \alpha)\)

But the variable is constrained by the appropriate edge from the environment
Coercions

Annotated terms are not primitive, but **syntactic sugar**

$$ (a : \sigma) \triangleq c_{\sigma} \ a $$

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

Coercion functions

The domain of the arrow is frozen

The codomain can be freely instantiated
Propagation

Used to enforce the constraints imposed by an instantiation edge

\[
g : \forall \alpha. \alpha \rightarrow (\beta \rightarrow \beta)
\]
\[
n : \forall \gamma. \gamma \rightarrow \gamma
\]
Propagation

Used to **enforce** the constraints imposed by an instantiation edge

We copy the type scheme

\[
g : \forall \alpha. \alpha \to (\beta \to \beta)
\]

\[
n : \forall \gamma. \gamma \to \gamma
\]
Propagation

Used to **enforce** the constraints imposed by an instantiation edge

We copy the type scheme, and add an unification edge between the constrained node and this copy

\[
g : \forall \alpha. \alpha \rightarrow (\beta \rightarrow \beta) \\
n : \forall \gamma. \gamma \rightarrow \gamma
\]
Propagation

Used to enforce the constraints imposed by an instantiation edge

We copy the type scheme, and add an unification edge between the constrained node and this copy

\[
g : \forall \alpha. \alpha \to (\beta \to \beta) \\
n : (\beta \to \beta) \to (\beta \to \beta)
\]
Acyclic constraints

Constraints can encode problems with polymorphic recursion

\[
\text{let rec } x = a \text{ in } b
\]

Restriction to constraints with an \textit{acyclic} dependency relation

\textbf{Dependency relation}

\( g \) depends on \( g' \) if either \( g' \xrightarrow{+} g \) or if \( g' \xrightarrow{} n \) with \( n \xrightarrow{+} g \)

Typing constraints are acyclic
Solving acyclic constraints

<table>
<thead>
<tr>
<th></th>
<th>Solving a constraint $\chi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Solve the initial unification edges</td>
</tr>
<tr>
<td>2.</td>
<td>Order the instantiation edges according to the dependency relation</td>
</tr>
<tr>
<td>3.</td>
<td>Propagate the first unsolved instantiation edge $e$, and solve the unification edges this operation has created. This solves $e$, and does not break already solved instantiation edges</td>
</tr>
<tr>
<td>4.</td>
<td>Iterate step 3 until all instantiation edge are solved</td>
</tr>
</tbody>
</table>
Solving acyclic constraints

Solving a constraint $\chi$
1. Solve the initial unification edges
2. Order the instantiation edges according to the dependency relation
3. Propagate the first unsolved instantiation edge $e$, and solve the unification edges this operation has created
   This solves $e$, and does not break already solved instantiation edges
4. Iterate step 3 until all instantiation edge are solved

Correctness
This algorithm computes a principal instance of $\chi$ in which all edges are solved
Complexity of inference

ML: type inference is $\text{DExp-Time}$ complete
(if types are not printed)

[McAllester 2003]: type inference in $O(kn(d + \alpha(kn)))$

- $k$ is the maximal size of type schemes
- $d$ is the maximal nesting of type schemes
Complexity of inference

ML: type inference is DExp-Time complete (if types are not printed)

[McAllester 2003]: type inference in $O(kn(d + \alpha(kn)))$

- $k$ is the maximal size of type schemes
- $d$ is the maximal nesting of type schemes

In ML, $d$ is the maximal left-nesting of let (i.e. let $x = (\text{let } y = \ldots \text{ in } \ldots) \text{ in } \ldots$)
Complexity of inference

ML : type inference is DExp-Time complete (if types are not printed)

[McAllester 2003] : type inference in $O(kn(d + \alpha(kn)))$

- $k$ is the maximal size of type schemes
- $d$ is the maximal nesting of type schemes

In ML$^F$, unification has the same complexity as in ML, but we introduce more type schemes

Still, $d$ is invariant by right-nesting of let

Complexity of ML$^F$ type inference

Under the hypothesis that programs are composed of a cascade of toplevel let declarations, type inference in ML$^F$ has linear complexity.
Summary

- Graphic constraints provide a new, simple, presentation of efficient ML type inference.
- Our framework is generic: it extends to $ML^F$ by changing only unification and the operation of taking a fresh instance of a scheme.
- We obtain optimal theoretical complexity, and excellent practical complexity.

Graphs can be used to explain type inference in a simple way.
Perspectives

- Solved constraints are translated into an explicit language xMLF (this ensures type soundness of the system)

- Graphic constraints should help explain and implement all the variants of MLF—including HML and FPH

The good tool for ML-like type systems

See http://gallium.inria.fr/~remy/mlf for other MLF-related material