A graphical presentation of MLF types with a linear-time unification algorithm

Didier Rémy, Boris Yakobowski

INRIA Rocquencourt
A brief presentation of MLF

[Le Botlan-Rémy, ICFP 2003]

[Le Botlan, 2003]
<table>
<thead>
<tr>
<th><strong>Why ML pstmt</strong>?</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>ML</strong></th>
<th><strong>System F</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer $\forall$</td>
<td>Inner (1st class) $\forall$ (Good)</td>
</tr>
<tr>
<td>$\forall \alpha \beta. (\alpha \to \beta) \to \alpha \ t \to \beta \ t$</td>
<td>$\lambda(f : \forall \alpha. \alpha \to \alpha)(f \ [\text{int}] \ 1, \ f \ [\text{bool}] \ 'b')$</td>
</tr>
<tr>
<td>Full type inference (Good)</td>
<td>Explicitly typed (undecidable type inference)</td>
</tr>
<tr>
<td></td>
<td>Fully annotated terms are (too) verbose</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow$ Need for partial type inference</td>
</tr>
</tbody>
</table>
MLF features

Conservative extension of both ML and System F

- ML programs need no annotations (type inference)
- F terms need fewer annotations
  type abstraction and applications are inferred
- Annotations are only required on λ-abstractions that are used polymorphically used
**ML\textsuperscript{F} features**

Conservative extension of both ML and System F

- **ML** programs need **no annotations** (type inference)
- **F** terms need **fewer annotations**
  - type abstraction and applications are inferred
- **Annotations** are only required on \(\lambda\)-abstractions that are used polymorphically used

**Principal types** (taking user-provided annotations into account)

Robust to small program transformations

e.g. if \(E[a_1 \ a_2]\) is typable so is \(E[\text{apply} \ a_1 \ a_2]\)

(where apply is \(\lambda f.\lambda x.f \ x\))
Example: type of choose id

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{id} = \lambda x.x )</td>
<td>( \forall \alpha. \alpha \to \alpha ) (( \tau_{id} ))</td>
</tr>
<tr>
<td>( \text{choose} = \lambda x.\lambda y.\text{if } b \text{ then } x \text{ else } y )</td>
<td>( \forall \gamma. \gamma \to \gamma \to \gamma )</td>
</tr>
</tbody>
</table>
Example: type of choose id

In System F, two different typings for choose id:

\[
\begin{align*}
\text{choose } [\forall \alpha \cdot \alpha \rightarrow \alpha] \text{ id} &: \quad \tau_{id} \rightarrow \tau_{id} \\
\Lambda \alpha \cdot \text{choose } [\alpha \rightarrow \alpha] \quad (\text{id } \alpha) &: \quad \forall \alpha \cdot (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)
\end{align*}
\]

$F_1$ 

$F_2$
Example: type of choose id

In System F, two different typings for choose id:

\[
\text{choose } [\forall \alpha \cdot \alpha \to \alpha] \text{ id } : \quad \tau_{id} \to \tau_{id} \quad F_1
\]

\[
\Lambda \alpha \cdot \text{choose } [\alpha \to \alpha] \quad (\text{id } \alpha) : \quad \forall \alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha) \quad F_2
\]

In MLF (note the absence of type annotations):

\[
\text{choose id : } \quad \forall (\beta = \tau_{id}) \beta \to \beta \quad \tau_1
\]

\[
: \forall (\alpha) \forall (\beta = \alpha \to \alpha) \beta \to \beta \quad \tau_2
\]

But \( \tau = \forall (\beta \geq \tau_{id}) \beta \to \beta \) is another, principal, typing:

\[
\tau \sqsubseteq \begin{cases} 
\forall (\beta \geq \text{int} \to \text{int}) \beta \to \beta \quad (\text{i.e. } (\text{int} \to \text{int}) \to (\text{int} \to \text{int})) \\
\forall (\beta = \forall (\eta = \tau_{id}) \eta \to \eta) \beta \to \beta \quad (\text{i.e. } (\tau_{id} \to \tau_{id}) \to (\tau_{id} \to \tau_{id})) \\
\tau_1, \tau_2
\end{cases}
\]
Syntactic presentation

A lot of administrative rules

- Hides the underlying principles
- Heavy proofs
- Makes extensions difficult

Is the instance relation the best within the framework?

Expensive unification (and hence type inference) algorithms.

Would it scale up to large or automatically generated programs?
Graphs are used instead of trees to represent types.

Graphs had already been proposed as a simpler representation, but were not formalized.

- Simpler presentation, strongly related to first-order types
- Proofs are shorter and simpler
- Unification has good complexity
Representing first and second-order types
Representing first-order types

\((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)\) as: a tree
Representing first-order types

$$(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$$ as: a tree a dag

All occurrences of a variable are shared.
Representing first-order types

\[(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)\] as: a tree an anonymous dag

Variables can be \(\alpha\)-converted and do not need to be named
Representing first-order types

\[(\alpha \to \beta) \to (\alpha \to \beta)\] as: an anonymous dag with sharing

Non-variable nodes may be also shared
Binders are represented with explicit ∀ nodes

\[
\text{int} \rightarrow (\forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta))
\]

Problem: commuting or instantiating binders change the structure of the type
With bindings edges, between a variable and the node where the variable is introduced.

\[ \text{int} \rightarrow (\forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \]

Commutation of binders comes for free!
MLF types, graphically
∀ (α = ∀ (β ≥ ⊥) ∀ (η = ∀ (δ ≥ ⊥) β → δ) ∀ (ε ≥ ⊥) η → ε) α → α

As a graphic type:
As a graphic type:
As a graphic type:

\[ \alpha = \forall \beta \geq \bot \quad \forall \eta = \forall \delta \geq \bot \beta \rightarrow \delta \quad \forall \epsilon \geq \bot \eta \rightarrow \epsilon \quad \alpha \rightarrow \alpha \]

...plus a binding tree...
\[
\forall (\alpha = \forall (\beta \geq \bot) \forall (\eta = \forall (\delta \geq \bot) \beta \rightarrow \delta) \forall (\epsilon \geq \bot) \eta \rightarrow \epsilon) \alpha \rightarrow \alpha
\]

As a graphic type:

...superposed
Well-formedness of graphic types

Syntactically:

\[ \forall (\alpha \geq \forall (\delta) \beta \rightarrow \beta) \quad \forall (\beta \geq \forall (\eta) \eta \rightarrow \delta) \]

\[ \alpha \rightarrow \alpha \]

- There is a mutual dependency between \( \alpha \) and \( \beta \)

\[ \Rightarrow \text{Not a MLF type} \]
Well-formedness of graphic types

Graphically:

- The binder of a node \( n \) must dominate \( n \) in all the mixed paths between \( n \) and the root \( \epsilon \).

- There is a path between \( \delta \) and \( \epsilon \) which does not contain \( \alpha \).

\[ \Rightarrow \text{This graph is not a type} \]
Instance between graphic types
Only four different transformations on graphic types:

- Grafting
- Merging
- Raising
- Weakening

change the structure of the type

change the binding tree of the type

Plus some permissions on nodes governing the set of transformations that can be applied to a node
Flags and permissions

- Transformations whose inverse can be unsound: allowed on flexible nodes

- Transformations whose inverse is sound, but that cannot be made implicit while retaining type inference: allowed on rigid nodes:

<table>
<thead>
<tr>
<th>Binding path</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq^*$</td>
<td>Flexible</td>
</tr>
<tr>
<td>$(\geq</td>
<td>=)^*=$</td>
</tr>
<tr>
<td>Others</td>
<td>Locked</td>
</tr>
</tbody>
</table>
Similar to the ML instance rule + generalization
\[ \forall \alpha.\tau \leq \forall \beta.\tau[\alpha/\tau'] \]

- Replaces a variable node by a type
- Irreversible transformation (the shape of the type changes), the node must be flexible
Partly similar to the ML instance $\forall \alpha \beta. \alpha \rightarrow \beta \leq \forall \alpha. \alpha \rightarrow \alpha$

Merges together two identical subgraphs bound on the same node with the same flag

The nodes must be flexible or rigid
Scope extrusion \((\tau \rightarrow (\forall \alpha. \tau') \leq \forall \alpha. \tau \rightarrow \tau', \alpha \text{ not free in } \tau)\)

Used to prove that the type \(\forall (\beta \geq \forall (\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta\) of choose id can be instantiated into \(\forall (\alpha) \forall (\beta \geq \alpha \rightarrow \alpha) \beta \rightarrow \beta\)

The node must be flexible or rigid
Forbids some (irreversible) transformations under a node

Used to require some polymorphism
Full example of instance
Full example of instance

Grafting
Full example of instance

Raising
Full example of instance
Full example of instance

Raising
Full example of instance
Weakening
Full example of instance
Full example of instance

Merging
Full example of instance
Full example of instance

Merging
Full example of instance

21(13)/??
Definition: The instance relation $\sqsubseteq$ is $(\sqsubseteq^G \cup \sqsubseteq^M \cup \sqsubseteq^R \cup \sqsubseteq^W)^*$

Commutation: $\sqsubseteq$ is equal to $\sqsubseteq^G ; \sqsubseteq^R ; \sqsubseteq^{MW}$ (\sqsubseteq^{MW} is (\sqsubseteq^M \cup \sqsubseteq^W)^*)

Drastically simplifies proofs and reasonings on instance derivations
Unification
Unification problem:

Given two types $\tau_1$ and $\tau_2$, find $\tau_u$ such that $\tau_1 \sqsubseteq \tau_u$ and $\tau_2 \sqsubseteq \tau_u$
The unification algorithm proceeds in three steps:

1: Computes the structure of \( \tau_u \), by performing first-order unification on the structure of \( \tau_1 \) and \( \tau_2 \).

Cost \( O(n) \) (or \( O(n\alpha(n)) \), depending on the algorithm).
The unification algorithm proceeds in three steps:

1: Computes the **structure** of $\tau_u$, by performing first-order unification on the structure of $\tau_1$ and $\tau_2$.

2: Computes the **binding tree** of $\tau_u$.

If the nodes $n_1, \ldots, n_k$ of $\tau_1$ and $\tau_2$ are merged into $n$ in $\tau_u$:

- The binding edges of $n_1, \ldots, n_k$ are raised until they are all bound at the same level.
- The flag for $n$ is the least permissive flag on $n_1, \ldots, n_k$.

**Cost $O(n)$**: a top down visit.

Quite involved step. Uses an amortized $O(1)$ algorithm for computing least-common ancestors.
The unification algorithm proceeds in three steps:

1: Computes the structure of $\tau_u$, by performing first-order unification on the structure of $\tau_1$ and $\tau_2$.

2: Computes the binding tree of $\tau_u$.

3: Checks the permissions for the merging operations performed in step 1.
   Cost $O(n)$, slightly involved visit of $\tau_1$, $\tau_2$ and $\tau_u$. 
Unification algorithm

- **Sound:** $\tau_u$ is always an instance of $\tau_1$ and $\tau_2$

- **Complete:**
  - always returns an unifier if one exists
  - the unifier returned is principal (i.e. more general for $\sqsubseteq$) than any other unifier.

Thus it computes all unifiers

- **Good complexity:** linear in $\max(|\tau_1|, |\tau_2|)$
  
  Extension to linear in $\min(|\tau_1|, |\tau_2|)$ in practice
Conclusion

- Simpler relations and proofs
- Presentation more semantic, thanks to permissions.
  - New (relaxed) instance relation.
  - Not easily transposable on syntactic types
- Good complexity for unification
Conclusion

- Simpler relations and proofs
- Presentation more semantic, thanks to permissions.
  - New (relaxed) instance relation.
  - Not easily transposable on syntactic types
- Good complexity for unification

Future works

- Revisit type inference using graphs
- Recursive types
- ...