Graphical types and constraints
Second-order polymorphism and inference

Who? Boris Yakobowski, under the supervision of Didier Rémy
Where? INRIA Rocquencourt, project Gallium
When? 17th December, 2008
Outline

1 Introduction: polymorphism in programming languages

2 Graphic types and MLF instance

3 Type inference through graphic constraints

4 A Church-style language for MLF

5 Conclusion
## Types in programs

### Context

- **Safety** of software
- **Expressivity** of programming languages
## Types in programs

### Context

- **Safety** of software
- **Expressivity** of programming languages

A key tool for this: **Typing**

Prevents the programmer from writing some forms of erroneous code

* e.g. `1 + "I am a string"`

(Of course, semantically incorrect code is still possible)
Types in programs

Context

Safety of software

Expressivity of programming languages

A key tool for this: Typing

Prevents the programmer from writing some forms of erroneous code
  e.g. \(1 + "I am a string"\)

(Of course, semantically incorrect code is still possible)

Static typing is important

```plaintext
if (...) then
    x := x+1;
else // rarely executed code
    print_string(x)
```
Type inference

The compiler *infers* the types of the expressions of the program

- Removes the need to write (often *redundant*) type annotations
  ```java
  Node n = new Node();
  ```
- Facilitates rapid prototyping
- Can infer types *more general* than the ones the programmer had in mind
Type inference issues

Which type should we give to functions admitting more than one possible type?

**Example:** finding the length of a list

```ml
let rec length = function
    | []         -> 0
    | _ :: q     -> 1 + length q

length: \{ int list \to int
           float list \to int \}
ML-style polymorphism

Functions no longer receive monomorphic types, but type schemes

\[ \text{sort: } \forall \alpha. \alpha \text{list} \to \alpha \text{list} \]

An alternative way of saying

“for any type \( \alpha \), sort has type \( \alpha \text{list} \to \alpha \text{list} \)”

The symbol \( \forall \) introduces universal quantification
ML-style polymorphism

Functions no longer receive monomorphic types, but type schemes

\[ \forall \alpha. \ alpha \ list \rightarrow \ alpha \ list \]

An alternative way of saying

“for any type \( \alpha \), sort has type \( \alpha \ list \rightarrow \alpha \ list \)”

The symbol \( \forall \) introduces universal quantification

---

**ML Polymorphism**

One of the key reasons of the success of ML as a language

**Full** type inference

(annotations are never needed in programs)

Sometimes a bit **limited**

universal quantification only in front of the type
Second-order polymorphism

Universal quantification under arrows is allowed

\[ \lambda(f) \ f \ (\lambda(x) \ x) : \forall \alpha. ((\forall \beta. \beta \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha \]

Many uses:
- Encoding existential types
- Polymorphic iterators over polymorphic structures
- State encapsulation \( \text{runST} :: \forall \alpha. (\forall \beta. \text{ST} \beta \alpha) \rightarrow \alpha \)
- ...
Second-order polymorphism

Universal quantification under arrows is allowed

\[ \lambda(f).f(\lambda(x)x) : \forall \alpha. ((\forall \beta. \beta \to \beta) \to \alpha) \to \alpha \]

Many uses:
- Encoding existential types
- Polymorphic iterators over polymorphic structures
- State encapsulation  \( \text{runST} :: \forall \alpha. (\forall \beta. \text{ST} \beta \alpha) \to \alpha \)
- ... 

We want at least the expressivity of System F

But type inference in System F is undecidable!
System F as a programming language

System F does not have principal types

Example:

\[
\begin{align*}
\text{id} & \triangleq \lambda (x)\ x & : & \forall \beta. \beta \rightarrow \beta \\
\text{choose} & \triangleq \lambda (x)\ \lambda (y)\ x & : & \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha 
\end{align*}
\]
System F as a programming language

System F does not have principal types

Example:

\[ \text{id} \triangleq \lambda(x) \, x : \forall \beta. \, \beta \to \beta \]

\[ \text{choose} \triangleq \lambda(x) \, \lambda(y) \, x : \forall \alpha. \, \alpha \to \alpha \to \alpha \]

choose id :

\[
\begin{align*}
\forall \beta. \, (\beta \to \beta) \to (\beta \to \beta) & \quad \alpha = \forall \beta. \, \beta \to \beta \\
\forall \gamma. \, (\gamma \to \gamma) \to (\gamma \to \gamma) & \quad \alpha = \gamma \to \gamma
\end{align*}
\]

No type is more general than the other

This is a fundamental limitation of System-F (and more generally of System-F types)
Flexible quantification

\[
\forall (\alpha \geq \tau) \tau':
\]

- Both \(\tau\) and \(\tau'\) can be instantiated inside \(\forall (\alpha \geq \tau) \tau'\)
- All occurrences of \(\alpha\) in \(\tau'\) must pick the same instance of \(\tau\)
Adding flexible quantification to types

Flexible quantification

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

- Both $\tau$ and $\tau'$ can be instantiated inside $\forall (\alpha \geq \tau) \tau'$
- All occurrences of $\alpha$ in $\tau'$ must pick the same instance of $\tau$

Example:

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

$\sqsubseteq (\forall \beta. \beta \rightarrow \beta) \rightarrow (\forall \beta. \beta \rightarrow \beta)$

or $\sqsubseteq \forall \gamma. (\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)$
Adding rigid quantification

- Flexible quantification solves the problem of principality
- But not the fact that type inference is undecidable
Adding rigid quantification

Flexible quantification solves the problem of principality

But not the fact that type inference is undecidable

Rigid quantification

Instance-bounded quantification, of the form $\forall (\alpha = \tau) \tau'$

$\tau$ cannot (really) be instantiated inside $\forall (\alpha = \tau) \tau'$

But $\forall (\alpha = \tau) \alpha \rightarrow \alpha$ and $\forall (\alpha = \tau) \forall (\alpha' = \tau) \alpha \rightarrow \alpha'$

are different as far as type inference is concerned
MLF as a type system

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

### Compared to ML
- The expressivity of second-order polymorphism is available
- All ML programs remain typable unchanged

### Compared to System F
- MLF has type inference
- Programs have principal types (given their type annotations)

Moreover:
- in practice, programs require very few type annotations
- typable programs are stable under a wide range of program transformations
How to improve MLF

Limitations

- Instance-bounded quantification makes equivalence and instance between types unwieldy
- Meta-theoretical results dense and non-modular
- Algorithmic inefficiency of type inference
- Not suitable for use in a typed compiler, by lack of a language to describe reduction

My work

- Use graphic types and constraints to improve the presentation
- Study efficient type inference
- Define an internal language for MLF
Outline

1. Introduction: polymorphism in programming languages

2. Graphic types and MLF instance

3. Type inference through graphic constraints

4. A Church-style language for MLF

5. Conclusion
Graphic types: an alternative representation of types

A graphic type

A term-dag, representing the skeleton of the type

Sharing is important, but only for variables

Variables are anonymous

\[
\begin{align*}
\alpha \to \beta \to \alpha \to \beta \\
\gamma \to \gamma \to \gamma \\
\end{align*}
\]
Graphic types: an alternative representation of types

A graphic type

A term-dag, representing the skeleton of the type

- Sharing is important, but only for variables
- Variables are anonymous

A binding tree, indicating where variables are bound

∀α. (∀β₁. α → β₁) → (∀β₂. α → β₂)
Graphic types: an alternative representation of types

A graphic type

A term-dag, representing the skeleton of the type
  Sharing is important, but only for variables
  Variables are anonymous

A binding tree, indicating where variables are bound

Some well-scopedness properties

\[(\forall \alpha_1. \alpha_1 \rightarrow \text{int}) \rightarrow (\alpha_2 \rightarrow \text{int})\]

Ill-scoped!
Graphic types: an alternative representation of types

A graphic type

- A term-dag, representing the skeleton of the type
  - Sharing is important, but only for variables
  - Variables are anonymous

- A binding tree, indicating where variables are bound

- Some well-scopedness properties

Advantages of graphic types:

- Commutation of binders, no $\alpha$-conversion, no useless quantification...
- Bring closer theory and implementation
- Same formalism for different systems: ML, System F, ML$^F$, $F_{\leq}$, ...
Graphic $ML^F$ types

- Two kinds of binding edges, for flexible and rigid quantification
- Non-variables nodes can be bound

\[ \forall (\alpha \geq \bot) \forall (\gamma = \forall (\beta \geq \bot) \alpha \to \beta) \gamma \to \gamma \]
Graphic ML^F types

- Two kind of binding edges, for flexible and rigid quantification
- Non-variables nodes can be bound
- Sharing of non-variable nodes becomes important

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

Possible type for \( \lambda(x) \) \( x \)

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

Incorrect for \( \lambda(x) \) \( x \)
### Instance on graphic $\text{ML}^F$ types

#### The instance relation $\sqsubseteq$

Four atomic operations on graphs:
Instance on graphic ML$^F$ types

The instance relation $\sqsubseteq$

Four atomic operations on graphs:

**Grafting**: replacing a variable by a closed type (variable substitution)
Instance on graphic $\text{ML}^F$ types

**The instance relation $\sqsubseteq$**

Four atomic operations on graphs:

- **Grafting**: replacing a variable by a closed type
  (variable substitution)

- **Merging**: fusing two identical subgraphs
  (correlates the two corresponding subtypes)
Instance on graphic $\text{MLF}$ types

The instance relation $\sqsubseteq$

Four atomic operations on graphs:

- **Grafting**: replacing a variable by a closed type (variable substitution)
- **Merging**: fusing two identical subgraphs (correlates the two corresponding subtypes)
- **Raising**: edge extrusion (removes the possibility to introduce universal quantification)
Instance on graphic $\text{ML}^F$ types

The instance relation $\sqsubseteq$

Four atomic operations on graphs:

- **Grafting**: replacing a variable by a closed type (variable substitution)
- **Merging**: fusing two identical subgraphs (correlates the two corresponding subtypes)
- **Raising**: edge extrusion (removes the possibility to introduce universal quantification)
- **Weakening**: turns a flexible edge into a rigid one (forbids further instantiation of the corresponding type)
Instance on graphic $\text{MLF}^F$ types

The instance relation $\sqsubseteq$

Four atomic operations on graphs:

- **Grafting**: replacing a variable by a closed type  
  (variable substitution)

- **Merging**: fusing two identical subgraphs  
  (correlates the two corresponding subtypes)

- **Raising**: edge extrusion  
  (removes the possibility to introduce universal quantification)

- **Weakening**: turns a flexible edge into a rigid one  
  (forbids further instantiation of the corresponding type)

A control of **permissions** rejecting some unsafe instances
Permissions on nodes

Some instances on types would be unsound

**Example:** \( e \triangleq \lambda (x : \forall \alpha. \forall \beta. \alpha \to \beta) \ x \)

Correct type for \( e \)
Permissions on nodes

Some instances on types would be unsound

**Example:** \( e \triangleq \lambda(x : \forall \alpha. \forall \beta. \alpha \rightarrow \beta) \ x \)

Correct type for \( e \) 

Incorrect type for \( e \): 

\( e (\lambda(y) \ y) \) would have type 

\( \forall \alpha. \forall \beta. \alpha \rightarrow \beta \)
Permissions on nodes

Some instances on types would be unsound

Nodes receive permissions according to the binding structure above and below them

Permissions are represented by colors

All forms of instance are forbidden on red nodes, as well as grafting on orange ones

This ensures type soundness
Unification on $\text{ML}^F$ graphic types

Unification on graphic types:

- Finds the most general type $\tau$ such that $\tau_1 \subseteq \tau$ and $\tau_2 \subseteq \tau$ 
- Or unifies two nodes in a certain type (more general)
Unification on $\text{ML}^F$ graphic types

Unification on graphic types:

- Finds the most general type $\tau$ such that $\tau_1 \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$
- Or unifies two nodes in a certain type (more general)

Unification algorithm

- First-order unification on the skeleton
- Minimal raising and weakening so that the binding trees match
- Control of permissions
Unification on $\text{ML}^F$ graphic types

Unification on graphic types:

- Finds the most general type $\tau$ such that $\tau_1 \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$
- Or unifies two nodes in a certain type (more general)

Unification algorithm

- First-order unification on the skeleton
- Minimal raising and weakening so that the binding trees match
- Control of permissions

Unification

- is principal on all useful problems
- has linear complexity
Outline

1. Introduction: polymorphism in programming languages
2. Graphic types and MLF instance
3. Type inference through graphic constraints
4. A Church-style language for MLF
5. Conclusion
Type inference in graphic ML$^F$

- **Constraints** are an elegant way to present type inference
  - Scale better to non-toy languages
  - More general than an algorithm

- **Graphic constraints** as an extension of graphic types

- Can be used to perform type inference on graphic types
  Permit type inference for ML, ML$^F$, and probably other systems
Graphic constraints

Graphic types extended with four new constructs

- Unification edges ➦-----◆
  Force two nodes to be equal

- Existential nodes
  “Floating” nodes, used only to introduce other constraints

- Generalization nodes  □

- Instantiation edges  -----➔

- Same instance relation as on graphic types

Meta-theoretical results can be reused unchanged
Type generalization

Type generalization is essential in ML$^F$, just as in ML

Gen nodes are used to promote types into type schemes

\[
g : \forall \alpha. \alpha \rightarrow \alpha
\]
Type generalization

Type generalization is essential in MLF, just as in ML.

Gen nodes are used to promote types into type schemes.

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

\( \alpha \) is free at the level of \( g' \)
Type generalization

Type generalization is essential in MLF, just as in ML.

Gen nodes are used to promote types into type schemes.

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

\(\alpha\) is free at the level of \(g'\).

Gen nodes also delimit generalization scopes.
Instantiation edges

Constrain a node to be an instance of a type scheme

$\beta \perp \Leftarrow G \Rightarrow G \Leftarrow e \Rightarrow n \Rightarrow \alpha$,

$e$ constrains $n$ to be an instance of $g$
Instantiation edges

Constrain a node to be an instance of a type scheme

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

\[ e \text{ is solved} \quad \text{(take } \beta = \alpha \text{)} \]

\[ e \text{ constrains } n \text{ to be an instance of } g \]
Constrain a node to be an instance of a type scheme

\[ g : \beta \to \alpha \]
\[ n : \alpha \to \alpha \]

\( e \) is not solved \((\beta \neq \alpha)\)

\( e \) constrains \( n \) to be an instance of \( g \)
A presolution of a constraint $\chi$ is an instance of $\chi$ in which all the instantiation and unification edges are solved.

Presolutions correspond to typing derivations, and are in correspondence with Church-style $\lambda$-terms.
Semantics of constraints

Presolutions

A *presolution* of a constraint $\chi$ is an instance of $\chi$ in which all the instantiation and unification edges are solved.

Presolutions correspond to typing derivations, and are in correspondence with Church-style $\lambda$-terms.

Solutions

A *solution* of a constraint is the type scheme represented by a presolutions of a constraint.
Typing constraints

Source language: (MLF only)

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

Source language:  

\[ a ::= x | \lambda(x) \ a | a \ a | \text{let } x = a \text{ in } a | (a : \tau) | \lambda(x : \tau) \ a \]  

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

\\[ \begin{array}{c} \\
\text{the blue arrows are constraint edges for the free variables of } a \\
\end{array} \]
Typing constraints

Source language: (MLF only)

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

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

\[ \begin{array}{c}
\text{ represents the typing constraint for } a \\
\text{ the blue arrows are constraint edges for the free variables of } a
\end{array} \]

One generalization scope by subexpression

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

Same typing constraints for ML and MLF

the superfluous gen nodes can be removed in ML

MLF constraints can be instantiated by the more general types of MLF
Typing constraint for an abstraction

\[ \lambda(x) \ a \xrightarrow{\sim} \ G \]

\( \lambda(x) \ a \) can receive type \( \alpha \rightarrow \beta \), provided

- \( \alpha \) is the (common) type of all the occurrences of \( x \) in \( a \)
- \( \beta \) is an instance of the type of \( a \).
Typing constraint for an application

\[ a \ b \Rightarrow G \]

\[ a \ b \] can receive type \( \beta \), provided there exists \( \alpha \) such that
- \( a \rightarrow \beta \) is an instance of the type of \( a \)
- \( \alpha \) is an instance of the type of \( b \)
Typing constraint for a let

As in ML

Each occurrence of $x$ in $b$ must have a (possibly different) instance of the type of $a$
Typing constraint for variables

The variable node is constrained by the appropriate edge from the typing environment.
Acyclic constraints

Constraints can encode problems with polymorphic recursion

```latex
\text{let rec } x = a \text{ in } b
```

Restriction to constraints with an **acyclic** dependency relation

**Dependency relation**

\( g \) depends on \( g' \) if \( g' \) is in the scope of \( g \), or if \( g' \) \( \overset{\cdash}{\longrightarrow} \) \( n \) with \( n \) in the scope of \( g \)

All typing constraints are acyclic
Solving acyclic constraints

Demo
Solving acyclic constraints

Demo

Principal presolutions and solutions
Complexity of type 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
Complexity of type 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 type 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.
Outline

1. Introduction: polymorphism in programming languages

2. Graphic types and MLF instance

3. Type inference through graphic constraints

4. A Church-style language for MLF

5. Conclusion
An explicit language for $\text{ML}^F$

Study subject reduction in $\text{ML}^F$

To be used inside a typed compiler

$\text{ML}^F$ types are more expressive than $F$ ones

System $F$ cannot be used as a target language

Need for a core, Church-style, language for $\text{ML}^F$, called $\times\text{ML}^F$
From System F to xMLF

xMLF generalizes System F

Types: \( \sigma ::= \bot \mid \forall (\alpha \geq \sigma) \sigma \mid \alpha \mid \sigma \rightarrow \sigma \)

Rigid quantification is only needed for type inference, and is inlined in xMLF

Terms: \( a ::= x \mid \lambda (x : \sigma) \ a \mid a \ a \mid \text{let } x = a \ \text{in } a \mid \Lambda (\alpha \geq \sigma) \ a \mid a[\varphi] \)

Typing rules are the same as in System F, except for type application

\[
\text{TApp} \quad \quad \Gamma \vdash a : \sigma \quad \Gamma \vdash \varphi : \sigma \leq \sigma' \quad \quad \frac{}{\Gamma \vdash a[\varphi] : \sigma'}
\]
Type computations

Instance is explicitly witnessed through the use of type computations

\[ \varphi ::= \varepsilon \mid \varphi; \varphi \mid \triangleright \sigma \mid \alpha \triangleleft \mid \forall (\geq \varphi) \mid \forall (\alpha \geq) \varphi \mid \& \mid \& \]
Type computations

Instance is explicitly witnessed through the use of type computations

\[ \varphi ::= \varepsilon \mid \varphi ; \varphi \mid \sigma \mid \alpha \triangleleft \mid \forall (\triangleright \varphi) \mid \forall (\alpha \triangleright) \varphi \mid \& \mid \varnothing \]

\begin{align*}
\text{Inst-Reflex} & \quad \Gamma \vdash \varepsilon : \sigma \leq \sigma \\
\text{Inst-Trans} & \quad \Gamma \vdash \varphi_1 : \sigma_1 \leq \sigma_2 \quad \Gamma \vdash \varphi_2 : \sigma_2 \leq \sigma_3 \\
\Gamma & \vdash \varphi_1 \triangleright \varphi_2 : \sigma_1 \leq \sigma_3 \\
\text{Inst-Bot} & \quad \Gamma \vdash \triangleright \sigma : \bot \leq \sigma
\end{align*}

\begin{align*}
\text{Inst-Hyp} & \quad \alpha \triangleright \sigma \in \Gamma \\
\Gamma & \vdash \alpha \triangleleft : \sigma \leq \alpha \\
\text{Inst-Inner} & \quad \Gamma \vdash \varphi : \sigma_1 \leq \sigma_2 \\
\Gamma & \vdash \forall (\triangleright \varphi) : \forall (\alpha \triangleright \sigma_1) \sigma \leq \forall (\alpha \triangleright \sigma_2) \sigma
\end{align*}

\begin{align*}
\text{Inst-Outer} & \quad \Gamma, \varphi : \alpha \triangleright \sigma \vdash \varphi : \sigma_1 \leq \sigma_2 \\
\Gamma & \vdash \forall (\alpha \triangleright) \varphi : \forall (\alpha \triangleright \sigma) \sigma_1 \leq \forall (\alpha \triangleright \sigma) \sigma_2
\end{align*}

\begin{align*}
\text{Inst-Quant-Elim} & \quad \Gamma \vdash \& : \forall (\alpha \triangleright \sigma) \sigma' \leq \sigma'\{\alpha \leftarrow \sigma\} \\
\text{Inst-Quant-Intro} & \quad \alpha \notin \text{ftv}(\sigma) \\
\Gamma & \vdash \varnothing : \sigma \leq \forall (\alpha \triangleright \bot) \sigma
\end{align*}
Example: back to choose id

\[
\text{choose} \triangleq \Lambda(\alpha \geq \bot) \lambda(x : \alpha) \lambda(y : \alpha) x : \forall (\alpha \geq \bot) \alpha \rightarrow \alpha \rightarrow \alpha
\]

\[
\text{id} \triangleq \Lambda(\beta \geq \bot) \lambda(x : \beta) x : \forall (\beta \geq \bot) \beta \rightarrow \beta
\]

To make choose id well-typed, we must choose a type into which \(\alpha\) must be instantiated
Example: back to choose id

\[ \text{choose} \triangleq \Lambda(\alpha \geq \bot) \lambda(x : \alpha) \lambda(y : \alpha) x : \forall(\alpha \geq \bot) \alpha \rightarrow \alpha \rightarrow \alpha \]

\[ \text{id} \triangleq \Lambda(\beta \geq \bot) \lambda(x : \beta) x : \forall(\beta \geq \bot) \beta \rightarrow \beta \]

To make choose id well-typed, we must choose a type into which \( \alpha \) must be instantiated

\[ e \triangleq \Lambda(\gamma \geq \sigma_{id}) (\text{choose}[\forall (\geq \gamma); \&]) (\text{id}[\gamma \downarrow]) : \forall(\gamma \geq \sigma_{id}) \gamma \rightarrow \gamma \]
Example: back to choose id

\[
\begin{align*}
\text{choose} & \triangleq \Lambda(\alpha \geq \bot) \lambda(x : \alpha) \lambda(y : \alpha) x : \forall (\alpha \geq \bot) \alpha \rightarrow \alpha \rightarrow \alpha \\
\text{id} & \triangleq \Lambda(\beta \geq \bot) \lambda(x : \beta) x : \forall (\beta \geq \bot) \beta \rightarrow \beta
\end{align*}
\]

To make choose id well-typed, we must choose a type into which \(\alpha\) must be instantiated

\[
e \triangleq \Lambda(\gamma \geq \sigma_{id}) \underbrace{(\text{choose}[\forall (\geq \bowtie \gamma); \&])}_{\gamma \rightarrow \gamma \rightarrow \gamma} \underbrace{(\text{id}[\gamma \triangleleft])}_{\gamma} : \forall (\gamma \geq \sigma_{id}) \gamma \rightarrow \gamma
\]

\[
\begin{cases}
\{ e[\&] \} : \sigma_{id} \rightarrow \sigma_{id} \\
\{ e[\&; \forall (\delta \geq) (\forall (\geq \forall (\geq \bowtie \delta); \&); \&)] : \forall (\delta \geq \bot) (\delta \rightarrow \delta) \rightarrow (\delta \rightarrow \delta) \}
\end{cases}
\]
Reducing expressions

Usual $\beta$-reduction

$$(\lambda(x : \tau) \ a_1) \ a_2 \rightarrow a_1 \{x \leftarrow a_2\}$$

let $x = a_2$ in $a_1$ $\rightarrow$ $a_1 \{x \leftarrow a_2\}$$
Reducing expressions

Usual $\beta$-reduction

6 specific rules to reduce type applications

\[(\lambda (x : \tau) \ a_1) \ a_2 \longrightarrow a_1\\{x \gets a_2}\\]

\[\text{let } x = a_2 \text{ in } a_1 \longrightarrow a_1\\{x \gets a_2}\\]

\[a[\varepsilon] \longrightarrow a\\]
\[a[\varphi; \varphi'] \longrightarrow a[\varphi][\varphi']\\]
\[a[\bot] \longrightarrow \Lambda(\alpha \geq \bot) \ a\\
\text{if } \alpha \notin \text{ftv}(a)\\]

\[(\Lambda(\alpha \geq \tau) \ a)[\&] \longrightarrow a\\{\alpha \triangleleft \gets \varepsilon}\\{\alpha \gets \tau}\\]
\[(\Lambda(\alpha \geq \tau) \ a)[\forall (\geq \varphi)] \longrightarrow \Lambda(\alpha \geq \tau[\varphi]) \ a\\{\alpha \triangleleft \gets \varphi; \alpha \triangleleft}\\]
\[(\Lambda(\alpha \geq \tau) \ a)[\forall (\alpha \geq \varphi)] \longrightarrow \Lambda(\alpha \geq \tau) (a[\varphi])\\]
Reducing expressions

Usual $\beta$-reduction

6 specific rules to reduce type applications

Context rule

\[
(\lambda(x : \tau) \ a_1) \ a_2 \longrightarrow a_1\{x \leftarrow a_2\}
\]

\[
\text{let } x = a_2 \text{ in } a_1 \longrightarrow a_1\{x \leftarrow a_2\}
\]

\[
a[\varepsilon] \longrightarrow a
\]

\[
a[\varphi; \varphi'] \longrightarrow a[\varphi][\varphi']
\]

\[
a[\varepsilon] \longrightarrow \Lambda(\alpha \geq \bot) \ a
\]

if $\alpha \notin \text{ftv}(a)$

\[
(\Lambda(\alpha \geq \tau) \ a)[\&] \longrightarrow a\{\alpha \leftarrow \varepsilon\}\{\alpha \leftarrow \tau\}
\]

\[
(\Lambda(\alpha \geq \tau) \ a)[\forall (\geq \varphi)] \longrightarrow \Lambda(\alpha \geq \tau[\varphi]) \ a\{\alpha \leftarrow \varphi; \alpha \leftarrow\}
\]

\[
(\Lambda(\alpha \geq \tau) \ a)[\forall (\alpha \geq) \ varphi] \longrightarrow \Lambda(\alpha \geq \tau) (a[\varphi])
\]

\[
E\{a\} \longrightarrow E\{a'\}
\]

if $a \longrightarrow a'$
Results on $x\text{MLF}$

Correctness:

- **Subject reduction**, for all contexts (including under $\lambda$ and $\Lambda$)

- Progress for **call-by-value** with or without the value restriction, and for **call-by-name**

  This is the first time that MLF is proven sound for call-by-name

- Mechanized proof of a previous version of the system
Results on xMLF

Correctness:

- **Subject reduction**, for all contexts (including under $\lambda$ and $\Lambda$)

- Progress for **call-by-value** with or without the value restriction, and for **call-by-name**

  This is the first time that MLF is proven sound for call-by-name

- **Mechanized proof of a previous version of the system**

- **Confluence** of strong reduction

- **The reduction rule of System F for type applications is derivable**

\[(\Lambda(\alpha) \ a)[\sigma] \rightarrow a\{\alpha \leftarrow \sigma\}\]

 (when $a$ is a System F term, and $\sigma$ a System F type)
From presolutions to $\mathbf{xMLF}$ terms

$\mathbf{MLF}$ presolutions can be algorithmically translated into well-typed $\mathbf{xMLF}$ terms

This ensures the type soundness of our type inference framework
From presolutions to $\times\text{MLF}$ terms

- $\text{MLF}$ presolutions can be algorithmically translated into well-typed $\times\text{MLF}$ terms
  
  This ensures the type soundness of our type inference framework

- Nodes flexibly bound on gen nodes are translated into $\times\text{MLF}$ type abstractions

- The fact that an instantiation edge is solved is translated into a type computation
From presolutions to $\text{xML}^F$ terms: example

A presolution for $K \triangleq \lambda(x) \lambda(y) x$

$K : \forall (\alpha) \alpha \rightarrow \sigma_{id} \rightarrow \alpha$
From presolutions to $\text{XML}^F$ terms: example

A presolution for $K \triangleq \lambda(x) \lambda(y) \ x$

$K : \forall (\alpha) \alpha \rightarrow \sigma_{id} \rightarrow \alpha$

$\Lambda(\alpha) \ \lambda(x : \alpha) \ \underbrace{(\Lambda(\beta) \ \lambda(y : \beta) \ x)}_{\forall (\beta) \beta \rightarrow \alpha}$

$\alpha \rightarrow \sigma_{id} \rightarrow \alpha$
From presolutions to $\mathcal{MLF}$ terms: example

A presolution for $K \triangleq \lambda(x) \lambda(y) x$

$K : \forall (\alpha) \alpha \rightarrow \sigma_{id} \rightarrow \alpha$

\[
\Lambda(\alpha) \lambda(x : \alpha) \left( (\Lambda(\beta) \lambda(y : \beta) x) \left[ \forall (\geq \triangleright \sigma_{id}) ; \& \right] \right) \triangleq T(e)
\]

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

\[
\alpha \rightarrow \sigma_{id} \rightarrow \alpha
\]
Outline

1. Introduction: polymorphism in programming languages
2. Graphic types and MLF instance
3. Type inference through graphic constraints
4. A Church-style language for MLF
5. Conclusion
Related works

- Bringing **System F and ML closer**
  - restriction to predicative fragment
  - higher-order unification
  - local type inference
  - boxy types
  - FPH, HML

- **Typing constraints for ML**

- **Encoding MLF into System F**
Contributions

- Graphic types and constraints are the good way to study MLF
- Presentation of MLF well-understood, and modular
- Generic type inference framework: works indifferently for ML or MLF
- Optimal theoretical complexity, and excellent practical complexity for type inference

Graphs can be used to explain type inference in a simple way, and not only for MLF

- xMLF makes MLF suitable for use in a typed compiler
Perspectives

Extensions to advanced typing features

- qualified types
- GADTs, recursive types
- dependent types
- $F^\omega$

Revisit HML and FPH using our inference framework
Thanks
Equivalence and instance on types

- permits only more sharing/raising/weakening
  - exactly corresponds to implementation
  - simpler to reason about
Equivalence and instance on types

\( \sqsubseteq \) permits only more sharing/raising/weakening

- exactly corresponds to implementation
- simpler to reason about

\( \approx \) identifies monomorphic subparts represented differently
Equivalence and instance on types

\( \sqsubseteq \) permits only more sharing/raising/weakening
- exactly corresponds to implementation
- simpler to reason about

\( \approx \) identifies monomorphic subparts represented differently

\( \sqsubseteq \approx \) is \( \sqsubseteq \) modulo \( \approx \)
- monomorphic subparts need not be bound at all
- same expressivity as \( \sqsubseteq \)
Equivalence and instance on types

\( \sqsubseteq \) permits only more sharing/raising/weakening
- exactly corresponds to implementation
- simpler to reason about

\( \simeq \) identifies monomorphic subparts represented differently

\( \sqsubseteq \simeq \) is \( \sqsubseteq \) modulo \( \simeq \)
- monomorphic subparts need not be bound at all
- same expressivity as \( \sqsubseteq \)

\( \equiv \) views types up to rigid quantification and \( \simeq \)
Equivalence and instance on types

\(\sqsubseteq\) permits only more sharing/raising/weakening

- exactly corresponds to implementation
- simpler to reason about

\(\approx\) identifies monomorphic subparts represented differently

\(\sqsubseteq\approx\) is \(\sqsubseteq\) modulo \(\approx\)

- monomorphic subparts need not be bound at all
- same expressivity as \(\sqsubseteq\)

\(\equiv\) views types up to rigid quantification and \(\approx\)

\(\equiv\equiv\) is \(\equiv\) modulo \(\equiv\)

- most expressive system
- undecidable type inference
- terms typable for \(\equiv\equiv\) are typable for \(\sqsubseteq\) through type annotations
Expansion

Expansion takes a fresh instance of a type scheme
Expansion

Expansion takes a fresh instance of a type scheme

The structure of the type scheme is copied
Expansion

Expansion takes a fresh instance of a type scheme

The structure of the type scheme is copied

The nodes that are not local to the scheme are shared between the copy and the scheme
Expansion

Expansion takes a fresh instance of a type scheme

The structure of the type scheme is copied

The nodes that are not local to the scheme are shared between the copy and the scheme

Where to bind nodes?

- in ML^F, inner polymorphism
Expansion

Expansion takes a fresh instance of a type scheme $G \rightarrow \bot$.

The structure of the type scheme is copied.

The nodes that are not local to the scheme are shared between the copy and the scheme.

Where to bind nodes?
- in ML$^F$, inner polymorphism
- in ML, to the gen node at which the copy is bound (less general)
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

\[
G : \forall \alpha. \alpha \rightarrow (\beta \rightarrow \beta) \\
n : \forall \gamma. \gamma \rightarrow \gamma
\]
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 the 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 the copy

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

\[
n : (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)
\]

Solving the unification edges enforces the constraint
Coercions

Annotated terms are not primitive, but syntactic sugar

\[(a : \tau) \triangleq c_{\tau} a\]

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

Coercion functions

Primitives of the typing environment

The domain of the arrow is frozen

The codomain can be freely instantiated
Solving acyclic constraints

**Solving an acyclic constraint \( \chi \)**

1. Solve the initial unification edges (by unification)
2. **Order** the instantiation edges according to the dependency relation
3. **Propagate** the first **unsolved** instantiation edge \( e \), then **solve** the unification edges created
   This solves \( e \), and does not break the already solved instantiation edges
4. **Iterate** step 3 until all the instantiation edge are solved
Solving acyclic constraints

**Solving an acyclic constraint $\chi$**

1. Solve the initial unification edges (by unification)
2. **Order** the instantiation edges according to the dependency relation
3. **Propagate** the first unsolved instantiation edge $e$, then **solve** the unification edges created
   
   This solves $e$, and does not break the already solved instantiation edges
4. **Iterate** step 3 until all the instantiation edge are solved

**Correctness**

This algorithm computes a principal presolution of $\chi$