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. $1 + \texttt{"I am a string"}$

(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

```
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

```
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

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

$$\texttt{length}: \begin{cases} \texttt{int list} \rightarrow \texttt{int} \\ \texttt{float list} \rightarrow \texttt{int} \end{cases}$$

ML-style polymorphism

- Functions no longer receive monomorphic types, but type schemes

  $\texttt{sort}: \quad \forall\alpha.\, \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

- An alternative way of saying

  "for any type α, sort has type $\alpha\,\texttt{list} \to \alpha\,\texttt{list}$"

The symbol $\forall$ introduces universal quantification

ML-style polymorphism

- Functions no longer receive monomorphic types, but type schemes

  $\texttt{sort}: \quad \forall\alpha.\, \alpha\, \texttt{list} \to \alpha\, \texttt{list}$

- An alternative way of saying

  "for any type α, $\texttt{sort}$ has type $\alpha\, \texttt{list} \to \alpha\, \texttt{list}$"

  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

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

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

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 $\texttt{runST} :: \forall\alpha.\ (\forall\beta.\ \mathsf{ST}\ \beta\ \alpha) \to \alpha$
  - ...

- 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:

$$\begin{array}{rcll} \mathsf{id} & \triangleq & \lambda(x)\, x & : \quad \forall\beta.\ \beta \to \beta \\ \mathsf{choose} & \triangleq & \lambda(x)\, \lambda(y)\, x & : \quad \forall\alpha.\ \alpha \to \alpha \to \alpha \end{array}$$

System F as a programming language

- System F does not have principal types

Example:

$$\begin{array}{rcll} \mathsf{id} & \triangleq & \lambda(x)\, x & : \quad \forall\beta.\ \beta \to \beta \\ \mathsf{choose} & \triangleq & \lambda(x)\, \lambda(y)\, x & : \quad \forall\alpha.\ \alpha \to \alpha \to \alpha \end{array}$$

$$\mathsf{choose\ id} : \begin{cases} (\forall\beta.\ \beta \to \beta) \to (\forall\beta.\ \beta \to \beta) & \alpha = \forall\beta.\ \beta \to \beta \\ \forall\gamma.\ (\gamma \to \gamma) \to (\gamma \to \gamma) & \alpha = \gamma \to \gamma \end{cases}$$

No type is more general than the other

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

Adding flexible quantification to types

Flexible quantification

ML^{F} types extend System F types with an instance-bounded quantification of the form $\forall\,(\alpha \geqslant \tau)\;\tau'$:

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

Adding flexible quantification to types

Flexible quantification

ML$^{\mathsf{F}}$ types extend System F types with an instance-bounded quantification of the form $\forall\,(\alpha \geqslant \tau)\;\tau'$:

- Both τ and τ' can be instantiated inside $\forall\,(\alpha \geqslant \tau)\;\tau'$
- All occurrences of α in τ' must pick the same instance of τ
- **Example:**

$$\begin{array}{rcl} \text{choose id} & : & \forall\,(\alpha \geqslant \forall\beta.\ \beta \to \beta)\ \alpha \to \alpha \\ & \sqsubseteq & (\forall\beta.\ \beta \to \beta) \to (\forall\beta.\ \beta \to \beta) \\ \text{or} & \sqsubseteq & \forall\gamma.\ (\gamma \to \gamma) \to (\gamma \to \gamma) \end{array}$$

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 $\forall (\alpha = \tau)\, \tau'$

- τ cannot (really) be instantiated inside $\forall (\alpha = \tau)\, \tau'$
- But $\forall (\alpha = \tau)\, \alpha \to \alpha$ and $\forall (\alpha = \tau)\, \forall (\alpha' = \tau)\, \alpha \to \alpha'$
are different as far as type inference is concerned

ML^{F} 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

- ML^{F} 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 ML^{F}

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 ML^{F}

Outline

1 Introduction: polymorphism in programming languages

2 Graphic types and MLF instance

3 Type inference through graphic constraints

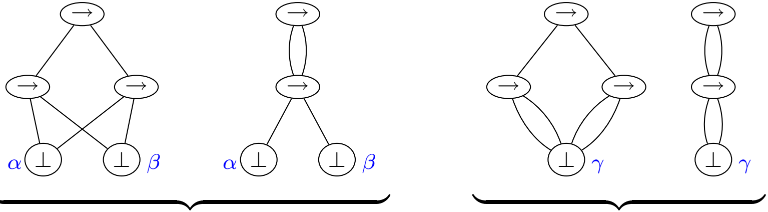
4 A Church-style language for MLF

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

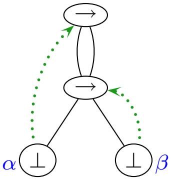


$(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\beta)$ $(\gamma\rightarrow\gamma)\rightarrow(\gamma\rightarrow\gamma)$

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

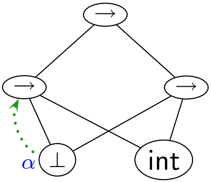


$$\forall\alpha.\ (\forall\beta_1.\ \alpha \rightarrow \beta_1) \rightarrow (\forall\beta_2.\ \alpha \rightarrow \beta_2))$$

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



$$(\forall \alpha_1.\ \alpha_1 \rightarrow \text{int}) \rightarrow (\boxed{\alpha_2} \rightarrow \text{int})$$

Ill-scoped!

Graphic types: an alternative representation of types

A graphic type

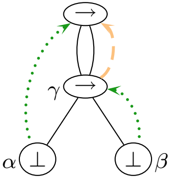
- A term-dag, representing the skeleton of the type
  - Sharing is important, but only for variables
  - Variables are anonymous
- A binding tree, indicating where variables are bound
- Some well-scopedness properties

Advantages of graphic types:

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

Graphic MLF types

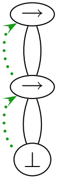
- Two kind of binding edges, for flexible and rigid quantification
- Non-variables nodes can be bound



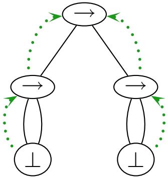
$$\forall\,(\alpha \geqslant \bot)\,\forall\,(\gamma = \forall\,(\beta \geqslant \bot)\;\alpha \to \beta)\;\gamma \to \gamma$$

Graphic MLF types

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



$\neq$



$\forall\,(\alpha \geqslant \sigma_{id})\;\alpha \to \alpha$

$\forall\,(\alpha \geqslant \sigma_{id})\,\forall\,(\beta \geqslant \sigma_{id})\;\alpha \to \beta$

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

Incorrect for $\lambda(x)\ x$

Instance on graphic 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)

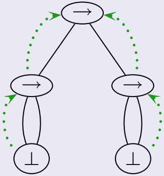
$\sqsubseteq$

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

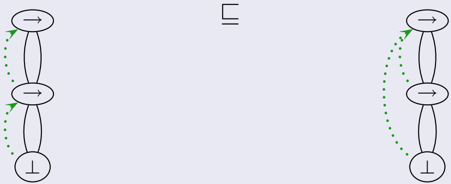


$\sqsubseteq$

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)
  - Merging: fusing two identical subgraphs (correlates the two corresponding subtypes)
  - Raising: edge extrusion (removes the possibility to introduce universal quantification)



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)
  - 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)

$\sqsubseteq$

Instance on graphic ML^F types

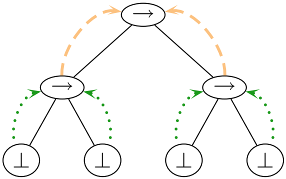
The instance relation $\sqsubseteq$

- Four atomic operations on graphs:
  - Grafting: replacing a variable by a closed type
    (variable substitution)
  - 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: $e \quad \triangleq \quad \lambda(x : \forall\alpha.\ \forall\beta.\ \alpha \to \beta)\ x$

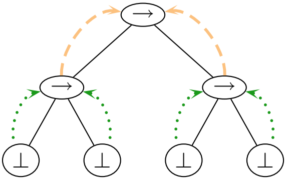


Correct type for e

Permissions on nodes

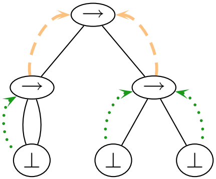
- Some instances on types would be unsound

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



Correct type for e

$\not\sqsubseteq$



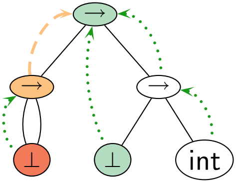
Incorrect type for e:

$e\ (\lambda(y)\ y)$ would have type
$\forall\alpha.\ \forall\beta.\ \alpha \to \beta$

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 ML^F graphic types

Unification on graphic types:

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

Unification on ML^F graphic types

Unification on graphic types:

- Finds the most general type τ 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 ML^F graphic types

Unification on graphic types:

- Finds the most general type τ 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 G
  - Instantiation edges
- Same instance relation as on graphic types

  Meta-theoretical results can be reused unchanged

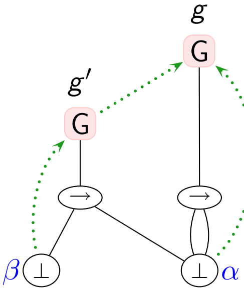
Type generalization

- Type generalization is essential in $\mathsf{ML^F}$, just as in ML
- Gen nodes are used to promote types into type schemes

$g : \quad \forall \alpha.\ \alpha \to \alpha$

Type generalization

- Type generalization is essential in ML^F, just as in ML
- Gen nodes are used to promote types into type schemes

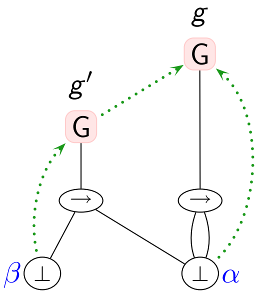


g : $\forall\alpha.\ \alpha \to \alpha$

g' : $\forall\beta.\ \beta \to \alpha$
α is free at the level of g'

Type generalization

- Type generalization is essential in ML^F, just as in ML
- Gen nodes are used to promote types into type schemes



$g: \quad \forall\alpha.\ \alpha \to \alpha$

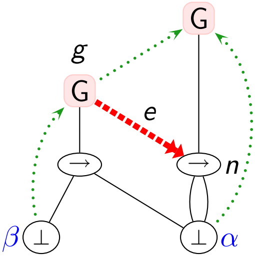
$g': \quad \forall\beta.\ \beta \to \alpha$

α is free at the level of g'

- Gen nodes also delimit generalization scopes

Instantiation edges

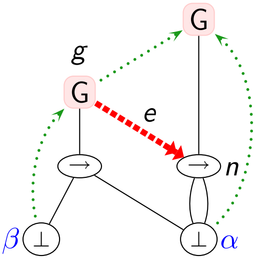
- Constrain a node to be an instance of a type scheme



- e constrains n to be an instance of g

Instantiation edges

- Constrain a node to be an instance of a type scheme



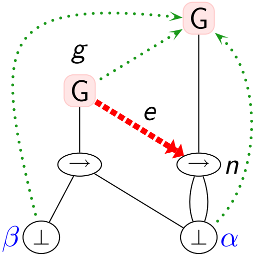
g : $\forall\beta.\ \beta \to \alpha$
n : $\alpha \to \alpha$

e is solved (take $\beta = \alpha$)

- e constrains n to be an instance of g

Instantiation edges

- Constrain a node to be an instance of a type scheme



$g: \quad \beta \to \alpha$
$n: \quad \alpha \to \alpha$

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

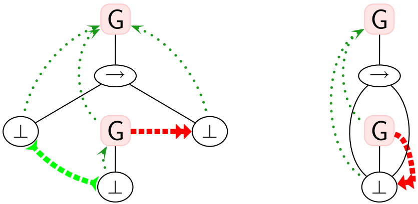
- *e* constrains *n* to be an instance of *g*

Semantics of constraints

Presolutions

A presolution of a constraint χ is an instance of χ in which all the instantiation and unification edges are solved.

Presolutions correspond to typing derivations, and are in correspondance with Church-style λ-terms



Semantics of constraints

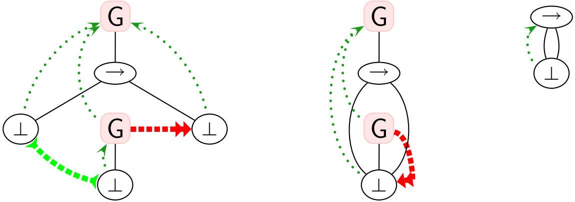
- ## Presolutions

  A presolution of a constraint χ is an instance of χ in which all the instantiation and unification edges are solved.

  Presolutions correspond to typing derivations, and are in correspondance with Church-style λ-terms

- ## Solutions

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



Typing constraints

- Source language: (ML^F 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: (ML^F only)

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

- λ-terms are translated into constraints compositionnally

  $\boxed{a}$ represents the typing constraint for a

  the blue arrows are constraint edges for the free variables of a

Typing constraints

- Source language: (ML^F only)

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

- λ-terms are translated into constraints compositionnally

  $\boxed{a}$ represents the typing constraint for a

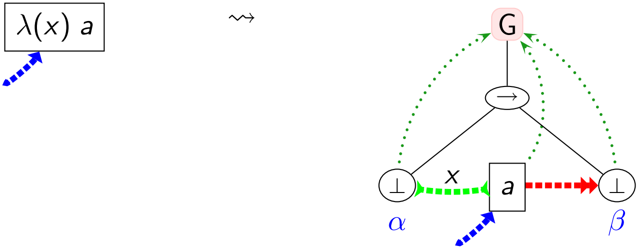
  the blue arrows are constraint edges for the free variables of a

- One generalization scope by subexpression

  in ML, only needed for let; in ML^F, needed everywhere

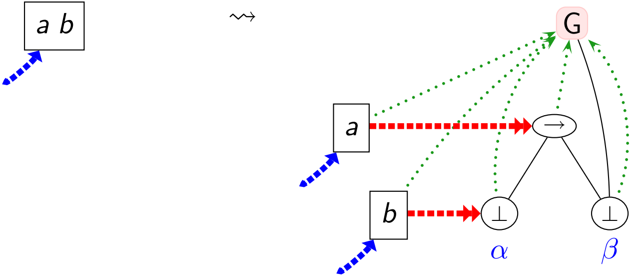
- Same typing constraints for ML and ML^F
  - the superfluous gen nodes can be removed in ML
  - ML^F constraints can be instantiated by the more general types of ML^F

Typing constraint for an abstraction



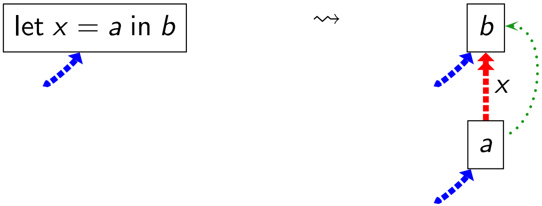
- $\lambda(x)\ a$ can receive type $\alpha \rightarrow \beta$, provided
  - α is the (common) type of all the occurrences of x in a
  - β is an instance of the type of a.

Typing constraint for an application



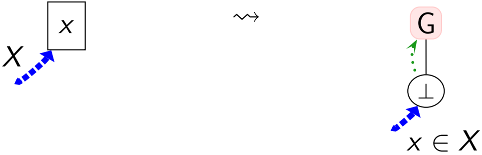
- *a b* can receive type β, provided there exists α such that
  - $a \to \beta$ is an instance of the type of *a*
  - α is an instance of the type of *b*

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

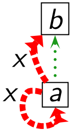


- the variable node is constrained by the appropriate edge from the typing environment

Acyclic constraints

- Constraints can encode problems with polymorphic recursion

$\boxed{\text{let rec } x = a \text{ in } b}$ $\rightsquigarrow$



- Restriction to constraints with an acyclic dependency relation

Dependency relation

g depends on g' if g' is in the scope of g, or if $g' \dashrightarrow n$ 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 $\mathrm{DExp\text{-}Time}$ 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 $\textsc{DExp-Time}$ 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 $x =$ (let $y = \ldots$ in $\ldots$) in $\ldots$)

Complexity of type inference

- ML : type inference is $\mathrm{DExp\text{-}Time}$ 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 langage for ML^{F}

- Study subject reduction in ML^{F}
- To be used inside a typed compiler

  ML^{F} types are more expressive than F ones

  System F cannot be used as a target langage
- Need for a core, Church-style, langage for ML^{F}, called $x\mathrm{ML}^{\mathsf{F}}$

From System F to $x\mathrm{ML}^\mathsf{F}$

$x\mathrm{ML}^\mathsf{F}$ generalizes System F

- Types: $\sigma \ ::= \ \bot \ \mid \ \forall\,(\alpha \geqslant \sigma)\,\sigma \ \mid \ \alpha \ \mid \ \sigma \to \sigma$

Rigid quantification is only needed for type inference, and is inlined in $x\mathrm{ML}^\mathsf{F}$

- Terms : $a \ ::= \ x \ \mid \ \lambda(x : \sigma)\ a \ \mid a\ a \ \mid \ \mathsf{let}\ x = a\ \mathsf{in}\ a$
  $\qquad \mid \ \Lambda(\alpha \geqslant \sigma)\ a \ \mid \ a[\varphi]$

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

$$\textsc{TApp}\quad \frac{\Gamma \vdash a : \sigma \qquad \Gamma \vdash \varphi : \sigma \leq \sigma'}{\Gamma \vdash a[\varphi] : \sigma'}$$

Type computations

Instance is explicitely witnessed through the use of type computations

$$\varphi \ ::= \ \varepsilon \ \mid \ \varphi ; \varphi \ \mid \ \triangleright \sigma \ \mid \ \alpha \triangleleft \ \mid \ \forall (\geqslant \varphi) \mid \ \forall (\alpha \geqslant) \, \varphi \ \mid \ \& \ \mid \ \invamp$$

Type computations

Instance is explicitely witnessed through the use of type computations

$$\varphi \;::=\; \varepsilon \;\mid\; \varphi ; \varphi \;\mid\; \triangleright \sigma \;\mid\; \alpha \triangleleft \;\mid\; \forall (\geqslant \varphi) \mid \forall (\alpha \geqslant) \, \varphi \;\mid\; \& \;\mid\; \aleph$$

Inst-Reflex

$$\frac{}{\Gamma \vdash \varepsilon : \sigma \leq \sigma}$$

Inst-Trans

$$\frac{\Gamma \vdash \varphi_1 : \sigma_1 \leq \sigma_2 \qquad \Gamma \vdash \varphi_2 : \sigma_2 \leq \sigma_3}{\Gamma \vdash \varphi_1 ; \varphi_2 : \sigma_1 \leq \sigma_3}$$

Inst-Bot

$$\frac{}{\Gamma \vdash \triangleright \sigma : \bot \leq \sigma}$$

Inst-Hyp

$$\frac{\alpha \geqslant \sigma \in \Gamma}{\Gamma \vdash \alpha \triangleleft : \sigma \leq \alpha}$$

Inst-Inner

$$\frac{\Gamma \vdash \varphi : \sigma_1 \leq \sigma_2}{\Gamma \vdash \forall (\geqslant \varphi) : \forall (\alpha \geqslant \sigma_1) \, \sigma \leq \forall (\alpha \geqslant \sigma_2) \, \sigma}$$

Inst-Outer

$$\frac{\Gamma, \varphi : \alpha \geqslant \sigma \vdash \varphi : \sigma_1 \leq \sigma_2}{\Gamma \vdash \forall (\alpha \geqslant) \, \varphi : \forall (\alpha \geqslant \sigma) \, \sigma_1 \leq \forall (\alpha \geqslant \sigma) \, \sigma_2}$$

Inst-Quant-Elim

$$\frac{}{\Gamma \vdash \& : \forall (\alpha \geqslant \sigma) \, \sigma' \leq \sigma' \{\alpha \leftarrow \sigma\}}$$

Inst-Quant-Intro

$$\frac{\alpha \notin \mathsf{ftv}(\sigma)}{\Gamma \vdash \aleph : \sigma \leq \forall (\alpha \geqslant \bot) \, \sigma}$$

Example: back to choose id

$$\begin{array}{rcll} \text{choose} & \triangleq & \Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ \lambda(y : \alpha)\ x & : \ \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \alpha \\ \text{id} & \triangleq & \Lambda(\beta \geqslant \bot)\ \lambda(x : \beta)\ x & : \ \forall\,(\beta \geqslant \bot)\ \beta \to \beta \end{array}$$

- To make choose id well-typed, we must choose a type into which α must be instantiated

Example: back to choose id

$$\begin{array}{rcll} \text{choose} & \triangleq & \Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ \lambda(y : \alpha)\ x : & \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \alpha \\ \text{id} & \triangleq & \Lambda(\beta \geqslant \bot)\ \lambda(x : \beta)\ x \qquad : & \forall\,(\beta \geqslant \bot)\ \beta \to \beta \end{array}$$

- To make choose id well-typed, we must choose a type into which α must be instantiated
- $e \triangleq \Lambda(\gamma \geqslant \sigma_{id})\ \underbrace{(\text{choose}[\forall\,(\geqslant \triangleright \gamma); \&])}_{\gamma\to\gamma}\ \underbrace{(\text{id}[\gamma \triangleleft])}_{\gamma} : \forall\,(\gamma \geqslant \sigma_{id})\ \gamma \to \gamma$

Example: back to choose id

$$\begin{aligned} \text{choose} &\triangleq \Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ \lambda(y : \alpha)\ x : \forall(\alpha \geqslant \bot)\ \alpha \to \alpha \to \alpha \\ \text{id} &\triangleq \Lambda(\beta \geqslant \bot)\ \lambda(x : \beta)\ x : \forall(\beta \geqslant \bot)\ \beta \to \beta \end{aligned}$$

- To make choose id well-typed, we must choose a type into which α must be instantiated
- $e \triangleq \Lambda(\gamma \geqslant \sigma_{id})\ \underbrace{(\text{choose}[\forall(\geqslant \triangleright \gamma); \&])}_{\gamma\to\gamma}\ \underbrace{(\text{id}[\gamma \triangleleft])}_{\gamma} : \forall(\gamma \geqslant \sigma_{id})\ \gamma \to \gamma$
- $\begin{cases} e[\&] & : \sigma_{id} \to \sigma_{id} \\ e[\otimes; \forall(\delta \geqslant)\ (\forall(\geqslant \forall(\geqslant \triangleright \delta); \&); \&)] & : \forall(\delta \geqslant \bot)\ (\delta \to \delta) \to (\delta \to \delta) \end{cases}$

Reducing expressions

- Usual β-reduction

$$
\begin{array}{rcl}
(\lambda(x:\tau)\ a_1)\ a_2 & \longrightarrow & a_1\{x \leftarrow a_2\} \\
\text{let } x = a_2 \text{ in } a_1 & \longrightarrow & a_1\{x \leftarrow a_2\}
\end{array}
$$

Reducing expressions

- Usual β-reduction
- 6 specific rules to reduce type applications

$$
\begin{array}{rcl}
(\lambda(x : \tau)\ a_1)\ a_2 & \longrightarrow & a_1\{x \leftarrow a_2\} \\
\text{let } x = a_2 \text{ in } a_1 & \longrightarrow & a_1\{x \leftarrow a_2\} \\
\\
a[\varepsilon] & \longrightarrow & a \\
a[\varphi; \varphi'] & \longrightarrow & a[\varphi][\varphi'] \\
a[\otimes] & \longrightarrow & \Lambda(\alpha \geqslant \bot)\ a \\
& & \text{if } \alpha \notin \mathrm{ftv}(a) \\
\\
(\Lambda(\alpha \geqslant \tau)\ a)[\&] & \longrightarrow & a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \tau\} \\
(\Lambda(\alpha \geqslant \tau)\ a)[\forall\,(\geqslant \varphi)] & \longrightarrow & \Lambda(\alpha \geqslant \tau[\varphi])\ a\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} \\
(\Lambda(\alpha \geqslant \tau)\ a)[\forall\,(\alpha \geqslant)\ \varphi] & \longrightarrow & \Lambda(\alpha \geqslant \tau)\ (a[\varphi])
\end{array}
$$

Reducing expressions

- Usual β-reduction
- 6 specific rules to reduce type applications
- Context rule

$$
\begin{array}{rcl}
(\lambda(x:\tau)\ a_1)\ a_2 & \longrightarrow & a_1\{x \leftarrow a_2\} \\
\text{let } x = a_2 \text{ in } a_1 & \longrightarrow & a_1\{x \leftarrow a_2\} \\
\\
a[\varepsilon] & \longrightarrow & a \\
a[\varphi;\varphi'] & \longrightarrow & a[\varphi][\varphi'] \\
a[\otimes] & \longrightarrow & \Lambda(\alpha \geqslant \bot)\ a \\
 & & \text{if } \alpha \notin \mathrm{ftv}(a) \\
\\
(\Lambda(\alpha \geqslant \tau)\ a)[\&] & \longrightarrow & a\{\alpha\lhd \leftarrow \varepsilon\}\{\alpha \leftarrow \tau\} \\
(\Lambda(\alpha \geqslant \tau)\ a)[\forall\,(\geqslant \varphi)] & \longrightarrow & \Lambda(\alpha \geqslant \tau[\varphi])\ a\{\alpha\lhd \leftarrow \varphi;\alpha\lhd\} \\
(\Lambda(\alpha \geqslant \tau)\ a)[\forall\,(\alpha \geqslant)\ \varphi] & \longrightarrow & \Lambda(\alpha \geqslant \tau)\ (a[\varphi]) \\
\\
E\{a\} & \longrightarrow & E\{a'\} \\
 & & \text{if } a \longrightarrow a'
\end{array}
$$

Results on xMLF

Correctness:

- Subject reduction, for all contexts (including under λ and Λ)

- 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 xML^F

Correctness:

- Subject reduction, for all contexts (including under λ and Λ)
- Progress for call-by-value with or without the value restriction, and for call-by-name

  This is the first time that ML^F 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

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

(when a is a System F term, and σ a System F type)

From presolutions to xML^F terms

- ML^F presolutions can be algorithmically translated into well-typed xML^F terms

This ensures the type soundness of our type inference framework

From presolutions to xML^F terms

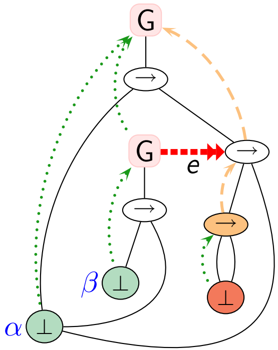
- ML^F presolutions can be algorithmically translated into well-typed xML^F terms

  This ensures the type soundness of our type inference framework

- Nodes flexibly bound on gen nodes are translated into xML^F type abstractions

- The fact that an instantiation edge is solved is translated into a type computation

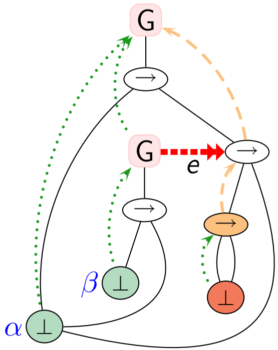
From presolutions to xML$^{\mathsf{F}}$ terms: example



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

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

From presolutions to xMLF terms: example

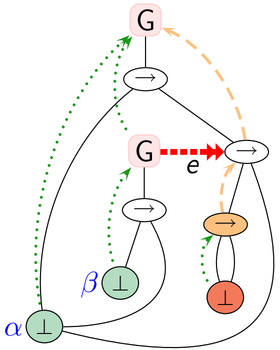


A presolution for $K \triangleq \lambda(x)\ \lambda(y)\ x$

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

$$\underbrace{\Lambda(\alpha)\ \ \lambda(x:\alpha)\ \ \underbrace{(\Lambda(\beta)\ \lambda(y:\beta)\ x)}_{\forall\,(\beta)\,\beta\to\alpha}}_{\alpha\to\sigma_{id}\to\alpha}$$

From presolutions to xML^F terms: example



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

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

$$\underbrace{\Lambda(\alpha)\ \ \lambda(x:\alpha)\ \ \underbrace{(\Lambda(\beta)\ \lambda(y:\beta)\ x)}_{\forall\,(\beta)\,\beta\to\alpha}\ \ \overbrace{[\forall\,(\geqslant \triangleright \sigma_{id});\&]}^{\mathcal{T}(e)}}_{\alpha\to\sigma_{id}\to\alpha}$$

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 ML^{F} into System F

Contributions

- Graphic types and constraints are the good way to study ML^{F}
- Presentation of ML^{F} well-understood, and modular
- Generic type inference framework: works indifferently for ML or ML^{F}
- 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 ML^{F}

- $x\mathrm{ML}^{\mathsf{F}}$ makes ML^{F} suitable for use in a typed compiler

Perspectives

- Extensions to advanced typing features
  - qualified types
  - GADTs, recursive types
  - dependent types
  - F^ω
- Revisit HML and FPH using our inference framework

Thanks

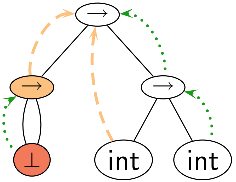
Equivalence and instance on types

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

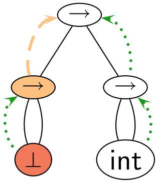
Equivalence and instance on types

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

- $\approx$ identifies monomorphic subparts represented differently



$\approx$

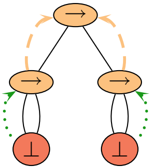


Equivalence and instance on types

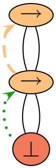
- $\sqsubseteq$ permits only more sharing/raising/weakening
  - exactly corresponds to implementation
  - simpler to reason about
- $\approx$ identifies monomorphic subparts represented differently
- $\sqsubseteq^{\approx}$ is $\sqsubseteq$ modulo $\approx$
  - monomorphic subparts need not be bound 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
- $\approx$ identifies monomorphic subparts represented differently
- $\sqsubseteq^{\approx}$ is $\sqsubseteq$ modulo $\approx$
  - monomorphic subparts need not be bound at all
  - same expressivity as $\sqsubseteq$
- $\boxminus$ views types up to rigid quantification and $\approx$



$\boxminus$

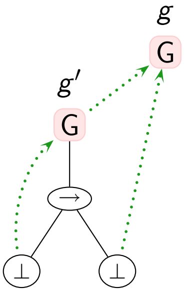


Equivalence and instance on types

- $\sqsubseteq$ 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 $\sqsubseteq$
- $\boxminus$ views types up to rigid quantification and $\approx$
- $\sqsubseteq^{\boxminus}$ is $\sqsubseteq$ modulo $\boxminus$
  - most expressive system
  - undecidable type inference
  - terms typable for $\sqsubseteq^{\boxminus}$ are typable for $\sqsubseteq$ through type annotations

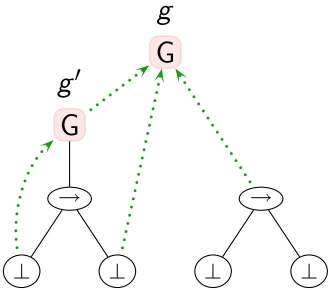
Expansion

Expansion takes a fresh instance of a type scheme



Expansion

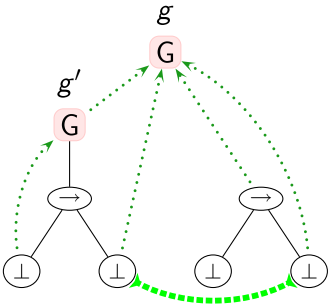
Expansion takes a fresh instance of a type scheme



- The structure of the type scheme is copied

Expansion

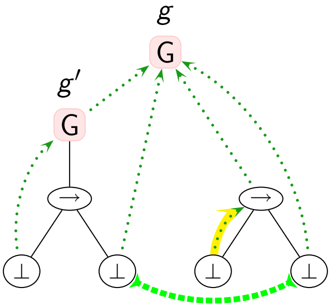
Expansion takes a fresh instance of a type scheme



- 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

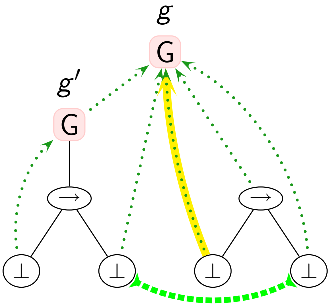


- The structure of the type scheme is copied
- The nodes that are not local to the scheme are shared between the copy and the scheme
- Where to bind nodes?

in ML^F, inner polymorphism

Expansion

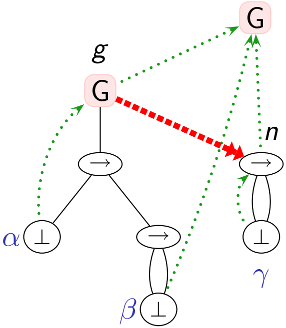
Expansion takes a fresh instance of a type scheme



- 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

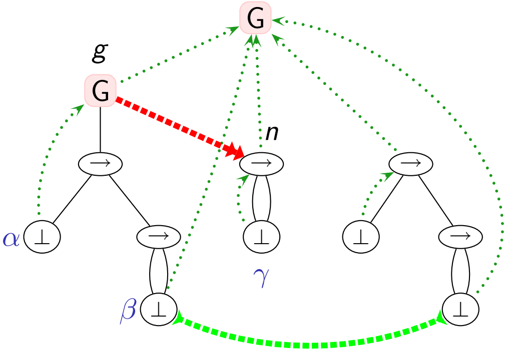
- Used to enforce the constraints imposed by an instantiation edge



g : $\forall\alpha.\ \alpha \to (\beta \to \beta)$
n : $\forall\gamma.\ \gamma \to \gamma$

Propagation

- Used to enforce the constraints imposed by an instantiation edge
- We copy the type scheme

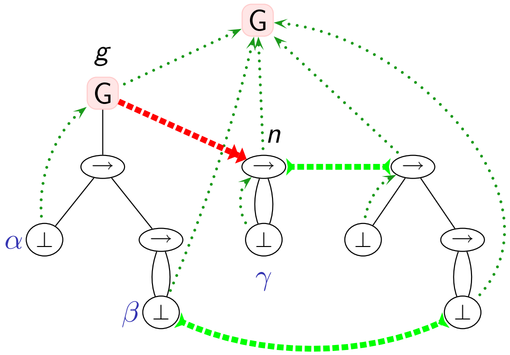


g : $\forall\alpha.\ \alpha \rightarrow (\beta \rightarrow \beta)$

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

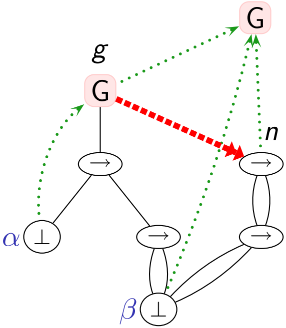


g : $\forall\alpha.\ \alpha \to (\beta \to \beta)$

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 the copy



$$g : \quad \forall \alpha.\ \alpha \to (\beta \to \beta)$$
$$n : \quad (\beta \to \beta) \to (\beta \to \beta)$$

- Solving the unification edges enforces the constraint

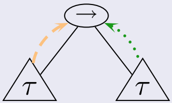
Coercions

- Annotated terms are not primitive, but syntactic sugar
  - $(a : \tau) \quad \triangleq \quad c_\tau \; a$
  - $\lambda(x : \tau)\; a \quad \triangleq \quad \lambda(x)\; \text{let } x = (x : \tau) \text{ in } a$

- **Coercion functions**

Primitives of the typing environment

c_τ :



- The domain of the arrow is frozen
- The codomain can be freely instantiated

Solving acyclic constraints

Solving an acyclic constraint χ

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 χ

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 χ