From ML to ML^F

Graphic type constraints with efficient type inference

Who? Boris Yakobowski, Didier Rémy

Where? INRIA, Gallium team

When? ICFP 2008

$\mathsf{ML^F}$

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

- $\mathsf{ML^F}$ 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

ML^{F} types

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

- All occurrences of α in τ' have a (same) instance of τ
- Both τ and τ' can be instantiated

$$
\begin{array}{lll}
\text{choose id} & : & \forall\,(\alpha \geqslant \forall\beta.\ \beta \to \beta)\ \alpha \to \alpha \\[1em]
 & \sqsubseteq & (\forall\beta.\ \beta \to \beta) \to (\forall\beta.\ \beta \to \beta) \\
 & & \text{taking } \alpha = \forall\beta.\ \beta \to \beta \\[1em]
 & \sqsubseteq & \forall\gamma.\ (\gamma \to \gamma) \to (\gamma \to \gamma) \\
 & & \text{taking } \alpha = \gamma \to \gamma \text{ for a fresh } \gamma
\end{array}
$$

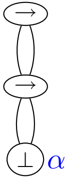
Graphic types

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

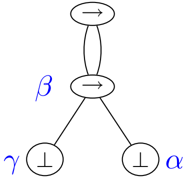
A graphic type

The superposition of

- a term-dag, representing the skeleton of the type



$\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$



$\forall\alpha.\ \forall(\beta \geqslant \forall\gamma.\ \gamma \to \alpha)\ \beta \to \beta$

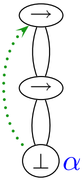
Graphic types

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

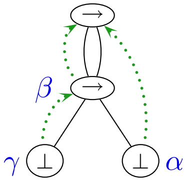
A graphic type

The superposition of

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



$\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$



$\forall\alpha.\ \forall(\beta \geqslant \forall\gamma.\ \gamma \to \alpha)\ \beta \to \beta$

Graphic constraints

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

Graphic constraints

- Used to perform ML or 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

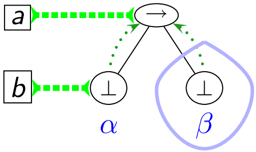
Graphic constraints

- Used to perform ML or ML^F type inference on graphic types
- An extension of graphic types (only three new constructs):
  - unification edges
  - generalization scopes
  - instantiation edges

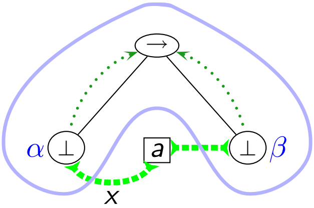
  Very small extension: we can reuse all the existing results on graphic types
- Using constraints is more general than a type inference algorithm

  e.g. different solving strategies

Typing abstractions or applications graphically



$$\mathcal{T}(a\ b) = \exists\alpha,\ \exists\beta,\\ (\alpha \to \beta = \mathcal{T}(a)\ \wedge\ \alpha = \mathcal{T}(b)).\ \beta$$

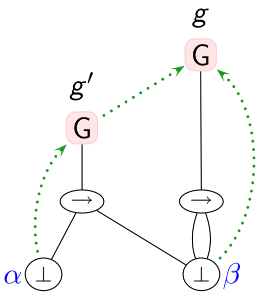


$$\mathcal{T}(\lambda(x)\ a) = \exists\alpha,\ \exists\beta,\\ (\alpha = \mathcal{T}(x)\ \wedge\ \beta = \mathcal{T}(a)).\ \alpha \to \beta$$

- 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

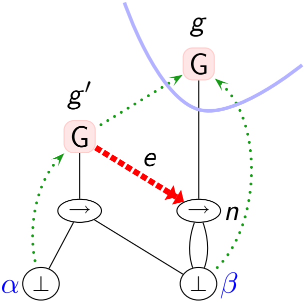


$$\begin{aligned} g &: \quad \forall\beta.\ \beta \to \beta \\ g' &: \quad \forall\alpha.\ \alpha \to \beta \\ & \quad\ \beta \text{ is free at the level of } g' \end{aligned}$$

- 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

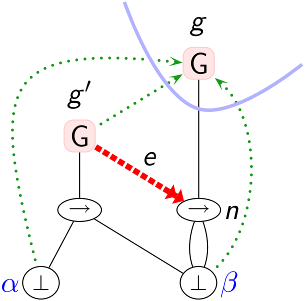


$$
\begin{aligned}
g' &: \quad \forall \alpha.\ \alpha \to \beta \\
n &: \quad \beta \to \beta
\end{aligned}
$$

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

Instantiation edge

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



$$\begin{aligned} g' &: \quad \alpha \rightarrow \beta \\ n &: \quad \beta \rightarrow \beta \end{aligned}$$

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

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)

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

- λ-terms are translated into typing constraints compositionnally

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

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

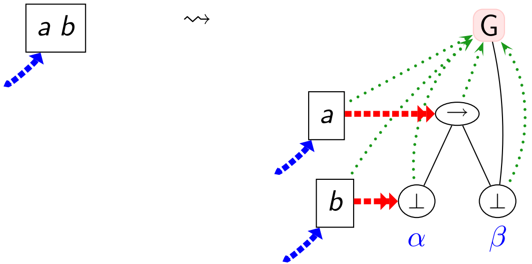
Typing constraints

- Source language: (ML^F only)

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

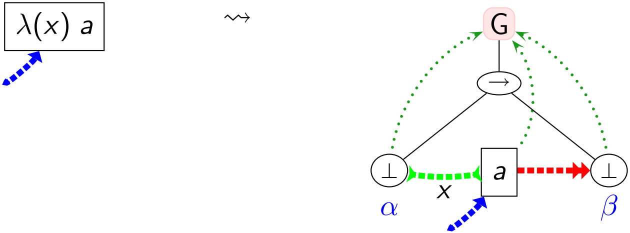
- λ-terms are translated into typing constraints compositionnally
- One generalization scope by subexpression
  in ML, only needed for let; in ML^F, needed everywhere
- Exact same typing constraints for ML and ML^F
  - the useless G-nodes can be removed in ML
  - ML^F constraints allow the more general types of ML^F, and have a more general notion of generalization

Typing constraint for an application



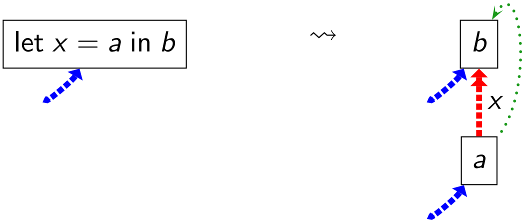
$$\mathcal{T}(a\ b) = \mathsf{GEN}(\exists\alpha,\ \exists\beta,\ (\mathcal{T}(a) \sqsubseteq \alpha \to \beta\ \wedge\ \mathcal{T}(b) \sqsubseteq \alpha).\ \beta)$$

Typing constraint for an abstraction



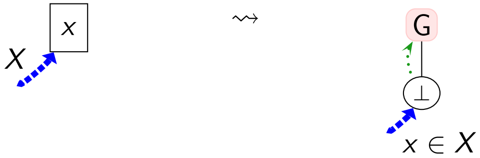
$$\mathcal{T}(\lambda(x)\ a) = \mathrm{GEN}(\exists\alpha,\ \exists\beta,\ (\mathcal{T}(x) = \alpha\ \wedge\ \mathcal{T}(a) \sqsubseteq \beta).\ \alpha \to \beta)$$

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



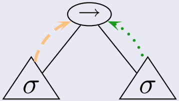
- A trivial type scheme $(\forall\alpha.\ \alpha)$
- But the variable is constrained by the appropriate edge from the environment

Coercions

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

- **Coercion functions**

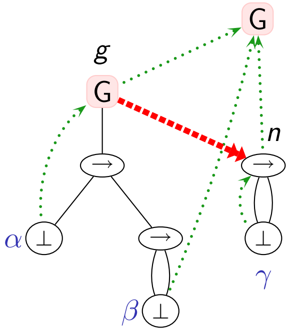
  c_σ :

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

Propagation

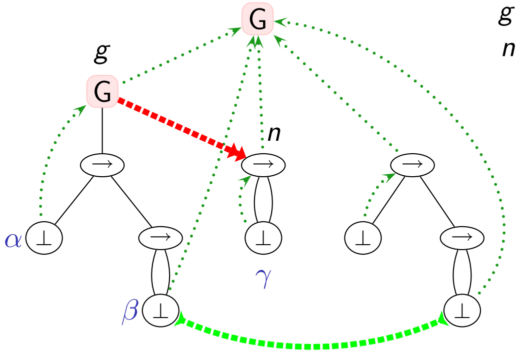
- Used to enforce the constraints imposed by an instantiation edge



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

Propagation

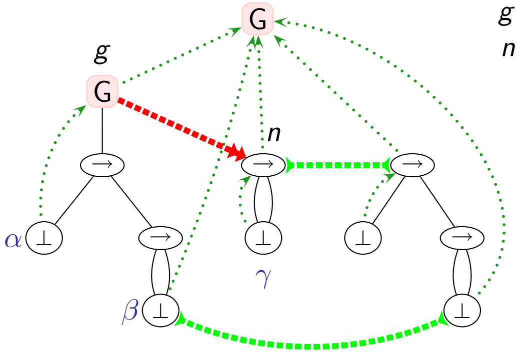
- Used to enforce the constraints imposed by an instantiation edge
- We copy the type scheme



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

Propagation

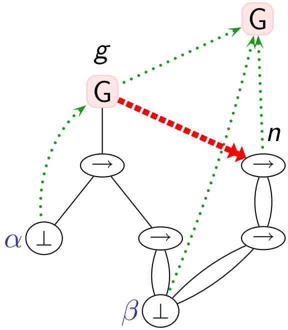
- Used to enforce the constraints imposed by an instantiation edge
- We copy the type scheme, and add an unification edge between the constrained node and this copy



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

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

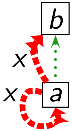


$$
\begin{aligned}
g &: \quad \forall\alpha.\ \alpha \to (\beta \to \beta) \\
n &: \quad (\beta \to \beta) \to (\beta \to \beta)
\end{aligned}
$$

Acyclic constraints

- Constraints can encode problems with polymorphic recursion

  let rec $x = a$ in b



- Restriction to constraints with an acyclic dependency relation

Dependency relation

g depends on g' if either $g' \overset{+}{\dashrightarrow} g$ or if $g' \twoheadrightarrow n$ with $n \overset{+}{\dashrightarrow} g$

- Typing constraints are acyclic

Solving acyclic constraints

Solving a constraint χ

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

Solving acyclic constraints

Solving a constraint χ

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 χ in which all edges are solved

Complexity of 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 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, d is the maximal left-nesting of let
  (*i.e.* let $x =$ (let $y = \ldots$ in $\ldots$) in $\ldots$)

Complexity of 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.

Summary

- Graphic constraints provide a new, simple, presentation of efficient ML type inference
- Our framework is generic: it extends to $\mathsf{ML^F}$ 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 $x\mathrm{ML}^{\mathsf{F}}$ (this ensures type soundness of the system)
- Graphic constraints should help explain and implement all the variants of ML^{F}—including HML and FPH

The good tool for ML-like type systems

See `http://gallium.inria.fr/~remy/mlf` for other ML^{F}-related material