Rich types, Tractable typing – Type Inference –

Yann Régis-Gianas
`yrg@irif.fr`

2017-12-08

Plan

Yann Régis-Gianas
yrg@irif.fr

What is type inference?

Yann Régis-Gianas
yrg@irif.fr

Informal definition

Type inference is a process that computes a type τ for a term t under some typing environment Γ if such a type exists. In other words, we are reading the judgment:

$$\Gamma \vdash t : \tau$$

where t and Γ are the inputs and τ is the output.

Hence, type inference determines if a term is typable.

Too vague!

Consider:

```
let f = fun x -> x + 1
```

Intuitively, a type inference engine must be able to process the term `x + 1` under an environment where the type of `x` is unknown.

Therefore, the typing environments used by a type inference engine differ slightly from the typing environments used by a type checker in the sense that the types bound to identifiers may contain pieces of unknown typing information. As soon as the syntax for types offers a notion of type variables, unknown typing information can be represented by free type variables.

This means that a free type variable occurring in a judgment may have two distinct roles: either it denotes a parameter of the typing derivation [1], or it denotes an unknown type to be instantiated by the inference algorithm. This distinction must be formally made.

[1] It is morally universally quantified at the meta-level.

Let us be more formal!

Let us write $\mathcal{V}$ to denote an infinite set of type variable identifiers from which we can generate fresh names. These type variables will denote the pieces of unknown typing information of the inference problem. The goal of type inference is to determine if these type variables can be assigned types to ensure the typability of the input term under the input environment.

Definition

A **type inference engine** is a partial function $\mathcal{I}$. This function expects $\mathcal{V}$, as well as a typing environment Γ and a term t. When defined, this function returns ϕ, an idempotent substitution whose domain is a finite subset of $\mathcal{V}$ and an inferred type τ.

Definition

A type inference engine $\mathcal{I}$ is **sound** if whenever $\mathcal{I}(\mathcal{V}, \Gamma, t) = (\mathcal{V}', \phi, \tau)$ then $\phi(\Gamma) \vdash t : \tau$.

The substitution ϕ witnesses the typability of t under Γ.

Yann Régis-Gianas
yrg@irif.fr

Parenthesis: α or $?\alpha$?

We could have followed another design choice by introducing a notion of **meta-variables**[2] to represent unknown types.

Yet, we will see that the strength of Hindley-Milner type system is to play with type variables to promote them from unknown types to parameters through the mechanism of **generalization**.

[2] As this is done in the Coq system for instance.

Soundness is not enough

Yann Régis-Gianas
yrg@irif.fr

As soon as a term is typable, the type inference algorithm must be able to find a typing for this term.

Definition

A type inference engine is **complete** and **principal** if whenever there exists a substitution ϕ such that $\phi(\Gamma) \vdash t : \phi(\tau)$ holds, then there exists ϕ', ϕ'' and $\mathcal{V}$ disjoint from $\mathsf{FTV}(\Gamma, \tau)$ such that:

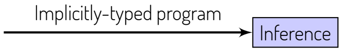
$$\begin{array}{rcl}
\mathcal{I}(\mathcal{V}, \Gamma, t) & = & (\mathcal{V}', \phi'', \tau') \\
\phi''(\tau') & = & \tau \\
\phi(\alpha) & = & (\phi'' \circ \phi')(\alpha) \quad \forall \alpha \notin \mathcal{V}
\end{array}$$

Yann Régis-Gianas
yrg@irif.fr

Type inference in a compiler

Type inference algorithms can be tricky to implement, in particular when we want them to be efficient. Fortunately, we have a safety net: the typechecker for the corresponding explicitly-typed language!

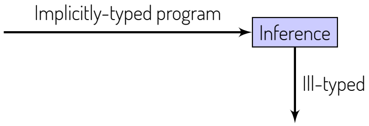
In practice, the so-called De Bruijn architecture is encouraged:



Type inference in a compiler

Type inference algorithms can be tricky to implement, in particular when we want them to be efficient. Fortunately, we have a safety net: the typechecker for the corresponding explicitly-typed language!

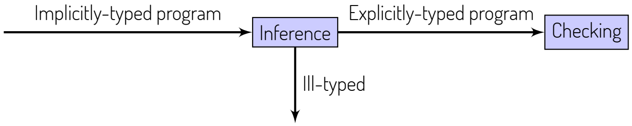
In practice, the so-called De Bruijn architecture is encouraged:



Type inference in a compiler

Type inference algorithms can be tricky to implement, in particular when we want them to be efficient. Fortunately, we have a safety net: the typechecker for the corresponding explicitly-typed language!

In practice, the so-called De Bruijn architecture is encouraged:



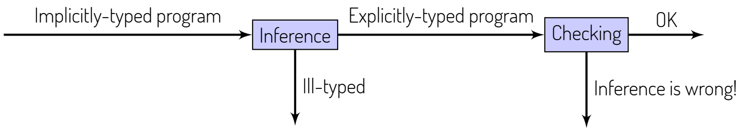
Yann Régis-Gianas
yrg@irif.fr

Yann Régis-Gianas
yrg@irif.fr

Type inference in a compiler

Type inference algorithms can be tricky to implement, in particular when we want them to be efficient. Fortunately, we have a safety net: the typechecker for the corresponding explicitly-typed language!

In practice, the so-called De Bruijn architecture is encouraged:

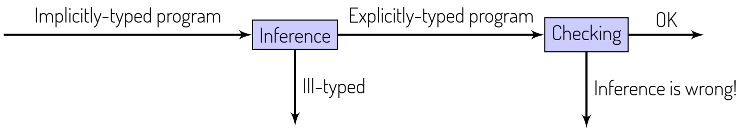


Type inference in a compiler

Yann Régis-Gianas
yrg@irif.fr

Type inference algorithms can be tricky to implement, in particular when we want them to be efficient. Fortunately, we have a safety net: the typechecker for the corresponding explicitly-typed language!

In practice, the so-called De Bruijn architecture is encouraged:



Elaboration

This extended type inference must **elaborate** an explicitly-typed program as a proof witness for the soundness of its answer. This proof is checked by an hopefully simpler, smaller and trustworthy program, the typechecker. Notice that the principality of the inferred program is not checked here.

From type inference to elaboration

Yann Régis-Gianas
yrg@irif.fr

Definition

A programming language is **implicitly typed** if its syntax allows the introduction of an identifier without declaring its type. The language is **explicitly typed** otherwise.

Let $\mathcal{L}_x$ be an explicitly typed language, and $\mathcal{L}_i$ be an implictly typed language sharing the same type-erasure semantics and the same type algebra.

Definition

A **sound elaboration engine** is a partial function $\mathcal{E}$ such that:

If $\mathcal{E}(\mathcal{V}, \Gamma, t) = (\mathcal{V}', \phi, \tau, t^\star)$ where $t \in \mathcal{L}_i$
then $t^\star \in \mathcal{L}_x$ and $\phi(\Gamma) \vdash t^\star : \phi(\tau)$

This course

- ML, syntax, semantics and type system(s)
- Algorithm $\mathcal{W}$ in OCaml
- Constraint-based approach
- Extensions of ML type inference

Plan

Yann Régis-Gianas
yrg@irif.fr

Plan

Yann Régis-Gianas
yrg@irif.fr

Yann Régis-Gianas
yrg@irif.fr

MiniML with constants

Syntax for terms (`Syntax.ITerm.t`)

$$
\begin{array}{rcl}
t & ::= & x \\
  & \mid & \lambda x.t \\
  & \mid & t\ t \\
  & \mid & \textbf{let}\ x = t\ \textbf{in}\ t
\end{array}
$$

Notice that the absence of type annotations on bindings.

Operational semantics

We assume a call-by-value weak reduction semantics.

A stratified type algebra

Syntax for types (**`Syntax.Type`**`.t`)

$$
\begin{array}{rcl}
\tau & ::= & \alpha \\
 & | & \varepsilon(\overline{\tau}) \\
\varepsilon & ::= & \rightarrow_2 \mid \textbf{int}_0 \mid \ldots
\end{array}
$$

A type is a first-order term.

Syntax for type schemes (**`Syntax.TypeScheme`**`.t`)

$$\sigma \quad ::= \quad \forall \overline{\alpha}.\tau$$

- The $\forall$ binder quantifies over (mono)types.
- Quantification is **prenex** : it cannot appear everywhere as in F.
- This is **predicative** rank-1 parametric polymorphism.

Instance relation

Definition

The type τ' is an instance of the type scheme $\forall\overline{\alpha}.\tau$, written $\forall\overline{\alpha}.\tau \preceq \tau'$, if there exists $\overline{\tau}$ such that $[\overline{\alpha} \mapsto \overline{\tau}]\tau = \tau'$.

Type system

Yann Régis-Gianas
yrg@irif.fr

The type system of ML is defined by two typing judgments:

$$\Gamma \vdash t : \tau \quad \text{and} \quad \Gamma \vdash t : \forall \overline{\alpha}.\tau$$

where $\Gamma ::= \bullet \mid \Gamma(x : \sigma)$.

While the first judgment is a standard typing judgment, the second can be seen as a family of standard typing judgments, parameterized by the types $\overline{\alpha}$.

Going from the second jugment to the first is an instanciation. The other way around is a generalization:

$$\text{Inst}\ \frac{\Gamma \vdash t : \sigma \qquad \sigma \preceq \tau}{\Gamma \vdash t : \tau} \qquad \text{Gen}\ \frac{\Gamma \vdash t : \tau \qquad \overline{\alpha} \mathbin{\#} \mathsf{FTV}(\Gamma)}{\Gamma \vdash t : \forall \overline{\alpha}.\tau}$$

Exercise

Yann Régis-Gianas
yrg@irif.fr

Exercise

Do you remember why the hypothesis

$$\overline{\alpha} \# \mathsf{FTV}(\Gamma)$$

is important in the Rule (Gen)?

Yann Régis-Gianas
yrg@irif.fr

Do you know ML?

Is

```
let f x = x in (f 0, f 'a')
```

equivalent to

```
(fun f -> (f 0, f 'a')) (fun x -> x)
```

?

Typing rules

In addition to (Gen), the rule (Var) can be used to introduce parameterized typing judgments:

$$\text{Var}\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

Binding type schemes to variables is the role of (Let):

$$\text{Let}\quad \frac{\Gamma \vdash t : \sigma \qquad \Gamma, (x : \sigma) \vdash u : \tau}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ u : \tau}$$

The rules for applications and abstractions are the same as for STLC:

$$\text{App}\quad \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\,u : \tau_2} \qquad \text{Abs}\quad \frac{\Gamma, (x : \forall\emptyset.\tau_1) \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$$

Yann Régis-Gianas
yrg@irif.fr

Properties of this type system

Yann Régis-Gianas
yrg@irif.fr

(We defer the discussion about the soundness of this type system.)

Question 1

Given an environment Γ and a typable term t, is there a unique type τ such that $\Gamma \vdash t : \tau$?

Question 2

Given a derivable judgment $\Gamma \vdash t : \tau$, is there a unique typing derivation that has this conclusion?

Properties of this type system

(We defer the discussion about the soundness of this type system.)

Question 1

Given an environment Γ and a typable term t, is there a unique type τ such that $\Gamma \vdash t : \tau$? **No!**

Question 2

Given a derivable judgment $\Gamma \vdash t : \tau$, is there a unique typing derivation that has this conclusion? **No! The rules are not syntax-directed.**

Properties of this type system

(We defer the discussion about the soundness of this type system.)

Question 1

Given an environment Γ and a typable term t, is there a unique type τ such that $\Gamma \vdash t : \tau$? **No!**

Question 2

Given a derivable judgment $\Gamma \vdash t : \tau$, is there a unique typing derivation that has this conclusion? **No! The rules are not syntax-directed.**

Question 1 will be tackled by the existence of principal type schemes.
Let us deal with Question 2 for now.

Yann Régis-Gianas
yrg@irif.fr

Syntax directed typing rules

We define another type system enjoying uniqueness of typing derivation, similar to the previous one except that the rules (Gen), (Inst), (Var) et (Let) are replaced by [3]:

Let-Gen
$$\frac{\Gamma \vdash t : \tau_1 \qquad \overline{\alpha} = \mathsf{FTV}(\tau_1) \setminus \mathsf{FTV}(\Gamma) \qquad \Gamma, (x : \forall \overline{\alpha}.\tau_1) \vdash u : \tau_2}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ u : \tau_2}$$

Var-Inst
$$\frac{\Gamma(x) \preceq \tau}{\Gamma \vdash x : \tau}$$

Since the system is now syntax-directed, does that mean that we have a type inference algorithm? or at least a type checking algorithm?

[3] We defer the proof of equivalence of the two type systems.

Syntax directed typing rules

Yann Régis-Gianas
yrg@irif.fr

We define another type system enjoying uniqueness of typing derivation, similar to the previous one except that the rules (Gen), (Inst), (Var) et (Let) are replaced by [3]:

Let-Gen
$$\frac{\Gamma \vdash t : \tau_1 \qquad \overline{\alpha} = \mathsf{FTV}(\tau_1) \setminus \mathsf{FTV}(\Gamma) \qquad \Gamma, (x : \forall \overline{\alpha}.\tau_1) \vdash u : \tau_2}{\Gamma \vdash \textbf{let } x = t \textbf{ in } u : \tau_2}$$

Var-Inst
$$\frac{\Gamma(x) \preceq \tau}{\Gamma \vdash x : \tau}$$

Since the system is now syntax-directed, does that mean that we have a type inference algorithm? or at least a type checking algorithm? Unfortunately, no. There remain too many choices:

- What are the types of λ-bound identifiers?
- How much generalization is needed?

$$\boxed{\text{Typability in ML is a non local property.}}$$

[3] We defer the proof of equivalence of the two type systems.

ML typing constraints

Yann Régis-Gianas
yrg@irif.fr

The non locality of the typability problem is better handled by unification constraints. Historically, these constraints are implicitly generated and solved on-the-fly by an algorithm called $\mathcal{W}$.

More recent approaches split type inference into two phases: a constraint generation and the solving of these contraints.

We will now implement and prove $\mathcal{W}$, turn it into an elaboration algorithm and present a constraint-based approach to ML type inference.

Plan

Yann Régis-Gianas
yrg@irif.fr

Do it yourself!

Yann Régis-Gianas
yrg@irif.fr

Programming exercise

1. Complete `Inference.algorithm_w`.
2. Fix the bug in `Syntax.Substitution.compose`.

Good old $\mathcal{W}$ of 82

$$\begin{array}{rcl}
\mathcal{W}(\mathcal{V}, \Gamma, x) & = & (\mathcal{V} \setminus \overline{\beta}, id, [\overline{\alpha} \mapsto \overline{\beta}]\tau) \\
\text{where} & & \Gamma(x) = \forall \overline{\alpha}.\tau \text{ and } \overline{\beta} \in \mathcal{V} \\
\\
\mathcal{W}(\mathcal{V}, \Gamma, \lambda x.t) & = & (\mathcal{V}', \phi, \phi(\alpha) \rightarrow \tau) \\
\text{where} & & \alpha \in \mathcal{V} \\
\text{and} & & \mathcal{W}(\mathcal{V} \setminus \alpha, \Gamma; (x : \alpha), t) = (\mathcal{V}', \phi, \tau) \\
\\
\mathcal{W}(\mathcal{V}, \Gamma, t\ u) & = & (\mathcal{V}'', \phi \circ \phi_t \circ \phi_u, \phi(\alpha)) \\
\text{where} & & \alpha \in \mathcal{V} \\
\text{and} & & \mathcal{W}(\mathcal{V} \setminus \alpha, \Gamma, u) = (\mathcal{V}', \phi_u, \tau_u) \\
\text{and} & & \mathcal{W}(\mathcal{V}', \phi_u(\Gamma), t) = (\mathcal{V}'', \phi_t, \tau_t) \\
\text{and} & & \phi = \mathsf{MGU}(\tau_t \overset{?}{=} \tau_u \rightarrow \alpha) \\
\\
\mathcal{W}(\mathcal{V}, \Gamma, \textbf{let } x = t \textbf{ in } u) & = & (\mathcal{V}'', \phi_2 \circ \phi_1, \tau_2) \\
\text{where} & & \mathcal{W}(\mathcal{V}, \Gamma, t) = (\mathcal{V}', \phi_1, \tau_1) \\
\text{and} & & \mathcal{W}(\mathcal{V}', \Gamma, (x : \sigma), u) = (\mathcal{V}'', \phi_2, \tau_2) \\
\text{and} & & \sigma = \mathsf{GEN}(\Gamma, \phi_1(\tau_1))
\end{array}$$

Yann Régis-Gianas
yrg@irif.fr

Auxiliary functions

Let us write U for a first-order unification problem made of a conjunction of equalities between types.

$$\begin{array}{rcl}
\mathsf{MGU}(x \overset{?}{=} x \wedge U) & = & \mathsf{MGU}(U) \\
\mathsf{MGU}(\tau \overset{?}{=} x \wedge U) & = & \mathsf{MGU}(x \overset{?}{=} \tau \wedge U) \\
 & \text{when} & \tau \text{ is not a variable} \\
\mathsf{MGU}(x \overset{?}{=} \tau \wedge U) & = & \mathsf{MGU}(U[x \mapsto \tau]) \circ [x \mapsto \tau] \\
 & \text{when} & \tau \text{ is not a variable and } x \notin \mathsf{FTV}(\tau) \\
\mathsf{MGU}(\varepsilon_1(\overline{\tau}_1) \overset{?}{=} \varepsilon_2(\overline{\tau}_2) \wedge U) & = & \mathsf{MGU}(\overline{\tau}_1 \overset{?}{=} \overline{\tau}_2 \wedge U) \\
 & \text{when} & \varepsilon_1 = \varepsilon_2 \\
\mathsf{MGU}(\top) & = & id
\end{array}$$

MGU is undefined for the other cases.

Besides, the generalization operation over types is defined as:

$$\mathsf{GEN}(\Gamma, \tau) = \forall(\mathsf{FTV}(\tau) \setminus \mathsf{FTV}(\Gamma)).\tau$$

Yann Régis-Gianas
yrg@irif.fr

Soundness

Yann Régis-Gianas
yrg@irif.fr

Theorem ($\mathcal{W}$ is sound)
If $\mathcal{W}(\mathcal{V}, \Gamma, t) = (\mathcal{V}', \phi, \tau)$ then $\phi(\Gamma) \vdash t : \tau$

Proof.
By induction over terms. (Details on the blackboard.) □

Completeness and principality

Yann Régis-Gianas
yrg@irif.fr

Theorem ($\mathcal{W}$ is complete and computes principal types)

If there exists ϕ and τ such that $\phi(\Gamma) \vdash t : \tau$,
then there exists $\mathcal{V}', \phi', \tau'$ and ρ such that

$$\begin{array}{rcll} \mathcal{W}(\mathcal{V}, \Gamma, t) & = & (\mathcal{V}', \phi', \tau') & \\ \tau & = & \rho(\tau') & \\ \phi(\alpha) & = & \phi'(\rho(\alpha)) & \quad \forall \alpha \notin \mathcal{V} \end{array}$$

Proof.
By induction over terms. (Details on the blackboard.) □

Completeness and principality

Yann Régis-Gianas
yrg@irif.fr

Theorem ($\mathcal{W}$ is complete and computes principal types)

If there exists ϕ and τ such that $\phi(\Gamma) \vdash t : \tau$,
then there exists $\mathcal{V}', \phi', \tau'$ and ρ such that

$$\begin{array}{rcll} \mathcal{W}(\mathcal{V}, \Gamma, t) & = & (\mathcal{V}', \phi', \tau') & \\ \tau & = & \rho(\tau') & \\ \phi(\alpha) & = & \phi'(\rho(\alpha)) & \quad \forall \alpha \notin \mathcal{V} \end{array}$$

Proof.
By induction over terms. (Details on the blackboard.) □

(In these proofs, the substitution manipulations are “tricky”!)

Back to Question 1

Yann Régis-Gianas
yrg@irif.fr

Question 1

Given an environment Γ and a typable term t, is there a unique type τ such that $\Gamma \vdash t : \tau$? **No!**

Yann Régis-Gianas
yrg@irif.fr

Back to Question 1

Question 1

Given an environment Γ and a typable term t, is there a unique type τ such that $\Gamma \vdash t : \tau$? **No!** But one can find a type that rules them all! $\mathcal{W}$ is a constructive proof of that fact!

Complexity of $\mathcal{W}$

Complexity

- ML typability is NP-hard and DEXPTIME-complete.
- Here is a typical example that requires an exponential time to type:

```
let f0 = fun x -> x in
let f1 = (f0, f0) in
let f2 = (f1, f1) in
...
fN
```

- **But** under reasonable assumptions[4], the complexity is quasi-linear.

[4]In the wild, the depth of types are bounded!

Plan

Yann Régis-Gianas
yrg@irif.fr

Do it yourself!

Programming exercise

1. Define the syntax of an explicitly typed version of ML.
2. Turn `Inference.algorithm_w` into an elaboration targeting the language you just defined.
3. Are you able to locate the binding site of every type variables that occur in the elaborated terms?
   (Let us name this question "Question 0".)

Yann Régis-Gianas
yrg@irif.fr

eML, an explicitly typed ML (Syntax)

We reuse the syntax of System F for abstractions with respect to types and for type applications:

$$
\begin{array}{rcl}
M & ::= & x \\
 & | & \lambda(x : \tau).M \\
 & | & M\ M \\
 & | & \Lambda\alpha.M \\
 & | & M\tau \\
 & | & \textbf{let}\ x : \sigma = M\ \textbf{in}\ M
\end{array}
$$

Notice that, contrary to System F, λ-bound identifiers are assigned a monomorphic type. As in ML, only **let**-bound identifiers can be polymorphic.

eML, an explicitly typed ML (Typing rules)

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, (x : \tau_1) \vdash M : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).M : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\, M_2 : \tau_2}$$

$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, (x : \sigma_1) \vdash M_2 : \sigma_2}{\Gamma \vdash \textbf{let}\ x : \sigma_1 = M_1\ \textbf{in}\ M_2 : \sigma_2} \qquad \frac{\Gamma, \alpha \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\tau : \sigma[\alpha \mapsto \tau]}$$

Wait a second!

$$\boxed{\text{But this is not the language we define in the previous exercise, right?}}$$

Yann Régis-Gianas
yrg@irif.fr

xML, another explicitly typed ML

Yann Régis-Gianas
yrg@irif.fr

The language we defined as a target of $\mathcal{W}$ elaboration is:

$$\begin{array}{rcl}
N & ::= & \Lambda\overline{\alpha}.Q \\
Q & ::= & x\overline{\tau} \\
 & \mid & \lambda(x : \tau).Q \\
 & \mid & Q\,Q \\
 & \mid & \textbf{let}\ x : \sigma = N\ \textbf{in}\ Q
\end{array}$$

xML, another explicitly typed ML

Yann Régis-Gianas
yrg@irif.fr

The language we defined as a target of $\mathcal{W}$ elaboration is:

$$
\begin{array}{rcl}
N & ::= & \Lambda\overline{\alpha}.Q \\
Q & ::= & x\overline{\tau} \\
 & \mid & \lambda(x:\tau).Q \\
 & \mid & Q\,Q \\
 & \mid & \textbf{let}\ x:\sigma = N\ \textbf{in}\ Q
\end{array}
$$

How do we relate eML and xML?

Yann Régis-Gianas
yrg@irif.fr

Normalization of eML typing derivations

Let us write

$$\Gamma \vdash M : \sigma \Rightarrow N$$

and

$$\Gamma \vdash M : \tau \Rightarrow Q$$

for two judgments that denote the normalization of the typing derivation of M as a typing derivation in xML.

More formally, we want the following properties to hold:

Lemma (Normalization preserves well-typedness)

If $\Gamma \vdash M : \sigma$ in eML and $\Gamma \vdash M : \sigma \Rightarrow N$ then $\Gamma \vdash N : \sigma$ in xML.
(Idem for the monomorphic case.)

Lemma (Well-formed typing derivations normalize)

If $\Gamma \vdash M : \sigma$ in eML then $\Gamma \vdash M : \sigma \Rightarrow N$.
(Idem for the monomorphic case.)

Do it yourself!

Yann Régis-Gianas
yrg@irif.fr

Formalization exercise

Define the rules for the previous two judgments.

Normalization rules

Let us start with

$$\Gamma \vdash M : \sigma \Rightarrow N$$

The syntax of N forces us to η-expand x:

Norm-Var

$$\frac{\Gamma(x) = \forall\overline{\alpha}.\tau}{\Gamma \vdash x : \forall\overline{\alpha}.\tau \Rightarrow \Lambda\overline{\alpha}.(x\,\overline{\alpha})}$$

The case for type abstraction is obvious:

Norm-TAbs

$$\frac{\Gamma, \alpha \vdash M : \sigma \Rightarrow N}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}$$

Yann Régis-Gianas
yrg@irif.fr

Normalization rules (continued)

Yann Régis-Gianas
yrg@irif.fr

Type applications are reduced during the normalization so that the resulting term N is normalized with respect to strong ι-reduction:

Norm-TApp
$$\frac{\Gamma \vdash M\tau : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}{\Gamma \vdash M\tau : \sigma[\alpha \mapsto \tau] \Rightarrow N[\alpha \mapsto \tau]}$$

To comply with the syntax, the type abstractions coming from the right-hand-side of **let**-bindings must be extruded:

Norm-Let
$$\frac{\Gamma \vdash M_1 : \sigma \Rightarrow N_1 \qquad \begin{array}{c}\overline{\alpha} \mathrel{\#} \sigma, N_1 \\ \Gamma, (x : \sigma) \vdash M_2 : \forall\overline{\alpha}.\tau \Rightarrow \Lambda\overline{\alpha}.Q\end{array}}{\Gamma \vdash \textbf{let}\ x : \sigma = M_1\ \textbf{in}\ M_2 : \forall\overline{\alpha}.\tau \Rightarrow \Lambda\overline{\alpha}.\textbf{let}\ x : \sigma = N_1\ \textbf{in}\ Q}$$

Normalization rules (continued)

Finally, the two rules for applications and λ-abstraction are straightforward since the two languages coincide on these constructions:

Norm-App
$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \Rightarrow Q_1 \qquad \Gamma \vdash M_2 : \tau_1 \Rightarrow Q_2}{\Gamma \vdash M_1 M_2 : \tau_2 \Rightarrow Q_1 Q_2}$$

Norm-Abs
$$\frac{\Gamma(x : \tau) \vdash M : \tau_2 \Rightarrow Q}{\Gamma \vdash \lambda(x : \tau).M : \tau_1 \to \tau_2 \Rightarrow \lambda(x : \tau).Q}$$

Back to Lemmas

Yann Régis-Gianas
yrg@irif.fr

Lemma (Well-formed typing derivations normalize)

If $\Gamma \vdash M : \sigma$ in eML then $\Gamma \vdash M : \sigma \Rightarrow N$.
(Idem for the monomorphic case.)

Proof.
Easy induction over typing derivations of eML. □

Lemma (Normalization preserves well-typedness)

If $\Gamma \vdash M : \sigma$ in eML and $\Gamma \vdash M : \sigma \Rightarrow N$ then $\Gamma \vdash N : \sigma$ in xML.
(Idem for the monomorphic case.)

Proof.
By induction over typing derivations of eML.
(Details on blackboard.) □

Two birds in one stone!

Lemma

Type erasure preservation If $\Gamma \vdash M : \sigma \Rightarrow N$ then the type erasures of M and N are equal.

Proof.

Immediate by induction. $\square$

Equivalence of two pairs of type systems for ML

- For any derivation of eML, there is an equivalent derivations of xML. (The other direction is obvious.) : We have a typechecker for explicitly-typed ML!
- If we remove the type annotations from the syntax, the proof can be transported to the (implicitly typed) ML type systems we have introduced earlier!
  (Question 2 is now solved.)

And what about “Question 0”?

Remember:

> Are you able to locate the binding site of every type variables that occur in the elaborated terms?

The unification type variables that have not been promoted to generalized type variables are still floating in the air. This is not really a problem: these variables can be seen as existentially quantified at the toplevel.

Yet, a cleaner treatment of these type variables consists in the introduction of an existential quantification over these (flexible) type variables at the level of terms:

$$t \quad ::= \quad \ldots \mid \exists \alpha . t$$

with a companion typing rule of the form:

$$\frac{\Gamma \vdash t : \tau[\alpha \mapsto \tau']}{\Gamma \vdash \exists \alpha . t : \tau}$$

Yann Régis-Gianas
yrg@irif.fr

An open question : the type soundness of ML

Many proofs are possible

- From scratch, by mimicking the proof for System F.
- By deducing the type soundness of eML from System F's.
  (See Didier Remy's course notes, section 4.6.3.)

Plan

Yann Régis-Gianas
yrg@irif.fr

To be continued...